Founded Semantics and Constraint Semantics of Logic Rules[*]

Yanhong A. Liu   Scott D. Stoller
Computer Science Department, Stony Brook University
{liu,stoller}@cs.stonybrook.edu

March 25, 2020

Abstract

Logic rules and inference are fundamental in computer science and have been studied extensively. However, prior semantics of logic languages can have subtle implications and can disagree significantly, on even very simple programs, including in attempting to solve the well-known Russell's paradox. These semantics are often non-intuitive and hard-to-understand when unrestricted negation is used in recursion.

This paper describes a simple new semantics for logic rules, *founded semantics*, and its straightforward extension to another simple new semantics, *constraint semantics*, that unify the core of different prior semantics. The new semantics support unrestricted negation, as well as unrestricted existential and universal quantifications. They are uniquely expressive and intuitive by allowing assumptions about the predicates, rules, and reasoning to be specified explicitly, as simple and precise binary choices. They are completely declarative and relate cleanly to prior semantics. In addition, founded semantics can be computed in linear time in the size of the ground program.

Keywords: Datalog, recursion, unrestricted negation, existential and universal quantifications, fixed-point semantics, constraints, well-founded semantics, stable model semantics.

1 Introduction

Logic rules and inference are fundamental in computer science, especially for solving complex modeling, reasoning, and analysis problems in critical areas, such as decision support, program analysis, verification, and security, and for knowledge representation and reasoning in general.

The semantics of logic rules and their efficient computations have been a subject of significant study, especially for complex rules that involve recursion and unrestricted negation and quantifications. Many different semantics and computation methods have been proposed, e.g., see surveys [AB94, Fit02]. Unfortunately, different semantics can disagree significantly, on even very simple programs. They are often non-intuitive and hard-to-understand when unrestricted negation is used in recursion. Even those used in many Prolog-based systems and Answer Set Programming systems—negation as failure [Cla78], well-founded semantics (WFS) [VRS91], and stable model semantics (SMS) [GL88]—can have subtle implications and differ significantly. Is it possible to create a simple semantics that also unifies these different semantics?

[*]This work was supported in part by NSF under grants CCF-1414078, CNS-1421893, IIS-1447549, CCF-1248184, CCF-0964196, and CCF-0613913, and ONR under grants N000142012751, N000141512208, and N000140910651.

In practice, different semantics may be useful under different assumptions about the facts, rules, and reasoning used. For example, an application may have complete information about some predicates, i.e., sets and relations, but not other predicates. Capturing such situations is important for increasingly larger and more complex applications. Any semantics that is based on a single set of assumptions for all predicates cannot best model such applications. How can a semantics be created to support all different assumptions and still be simple and easy to use?

This paper describes a simple new semantics for logic rules, *founded semantics*, and its straightforward extension to another simple new semantics, *constraint semantics*.

- The new semantics support unrestricted negation (both stratified and non-stratified), as well as unrestricted combinations of existential and universal quantifications.
- They allow each predicate to be specified explicitly as certain (each assertion of the predicate has one of two values: true, false) or uncertain (has one of three values: true, false, undefined), and as complete (all rules defining the predicate are given) or not.
- Completion rules are added for predicates that are complete, as explicit rules for inferring the negation of those predicates using the negation of the hypotheses of the given rules.
- Founded semantics infers all true and false values that are founded, i.e., rooted in the given true or false values and exactly following the rules, and it completes certain predicates with false values and completes uncertain predicates with undefined values.
- Constraint semantics extends founded semantics by allowing undefined values to take all combinations of true and false values that satisfy the constraints imposed by the rules.

Founded semantics and constraint semantics unify the core of previous semantics and have three main advantages:

1. They are expressive and intuitive, by allowing assumptions about predicates and rules to be specified explicitly, by including the choice of uncertain predicates to support common-sense reasoning with ignorance, and by adding explicit completion rules to define the negation of predicates.
2. They are completely declarative. Founded semantics takes the given rules and completion rules as recursive definitions of the predicates and their negation, and is simply the least fixed point of the recursive functions. Constraint semantics takes the given rules and completion rules as constraints, and is simply the set of all solutions that are consistent with founded semantics.
3. They relate cleanly to prior semantics, including stratified semantics [ABW88], first-order logic, Fitting semantics (also called Kripke-Kleene semantics) [Fit85], supported models [ABW88], as well as WFS and SMS, by precisely capturing corresponding assumptions about the predicates and rules.

Additionally, founded semantics can be computed in linear time in the size of the ground program, as opposed to quadratic time for WFS.

Finally, founded semantics and constraint semantics can be extended to allow uncertain, complete predicates to be specified as closed—making an assertion of the predicate false if inferring it to be true (respectively false) requires assuming itself to be true (respectively false)—and thus match WFS and SMS, respectively.

The rest of the paper is organized as follows. Section 2 gives informal motivation for the new semantics. Section 3 describes the rule language. Sections 4, 5, and 6 present the formal definition of the new semantics, properties of the semantics, and comparison with other semantics, respectively. Section 7 compares with other semantics for well-known small examples and more. Section 8 describes linear-time computation, closed predicate assumption, and other extensions. Section 9 discusses additional well-known examples, showing that we obtain the desired semantics for all of them. Section 10 discusses related work and concludes. Appendix A contains proofs of all theorems.

This paper is an extended and revised version of Liu and Stoller [LS18]. The main changes are new Theorem 3 in Section 5 to help simplify some results, new Section 7 on small examples that was in an appendix, new Section 9 that describes additional well-known examples, and new Appendix A that contains complete proofs of all nineteen theorems.

2 Motivation for founded semantics and constraint semantics

Founded semantics and constraint semantics are designed to be intuitive and expressive. For rules with no negation or with restricted negation, which have universally accepted semantics, the new semantics are consistent with the accepted semantics. For rules with unrestricted negation, which so far lack a universally accepted semantics, the new semantics unify the core of prior semantics with two basic principles:

1. Assumptions about certain and uncertain predicates, with true (T) and false (F) values, or possibly undefined (U) values, and about whether the rules defining each predicate are complete must be made explicit.

2. Any easy-to-understand semantics must be consistent with one where everything inferred that has a unique T or F value is rooted in the given T or F values and following the rules.

This section gives informal explanations.

Rules with no negation. Consider a set of rules with no negation in the hypotheses, e.g., a rule can be "`q(x) if p(x)`" but not "`q(x) if not p(x)`" for predicates `p` and `q` and variable `x`. The meaning of the rules, given a set of facts, e.g., a fact `p(a)` for constant `a`, is the set of all facts that are given or can be inferred by applying the rules to the facts, e.g., {`p(a)`,`q(a)`} using the example rule and fact given. In particular,

1. Everything is either T or F, i.e., T as given or inferred facts, or F as otherwise. So one can just explicitly express what are T, and the rest are F.

2. Everything inferred must be founded, i.e., rooted in the given facts and following the rules. So anything that always depends on itself, e.g., `p(a)`, given only the rule "`p(x) if p(x)`", is not T.

In technical terms, the semantics is *2-valued*, and the set of all facts, i.e., true assertions, is the *minimum model*, equal to the *least fixed point* of applying the rules starting from the given facts.

Rules with restricted negation. Consider rules with negation in the hypotheses, but with each negation only on a predicate all of whose facts can be inferred without using rules that contain negation of that predicate, e.g., one can have "`q(x) if not p(x)`" but not "`p(x) if not p(x)`". The meaning of the rules is as for rules with no negation except that a rule with negation is applied only after all facts of the negated predicates have been inferred. In other words,

> The true assertions of any predicate do not depend on the negation of that predicate. So a negation could be just a test after all facts of the negated predicate are inferred. The rest remains the same as for rules with no negation.

In technical terms, this is *stratified negation*; the semantics is still 2-valued, the minimum model, and the set of all true assertions is the least fixed point of applying the rules in order of the *strata*.

Rules with unrestricted negation. Consider rules with unrestricted negation in the hypotheses, where a predicate may cyclically depend on its own negation, e.g., "`p(x) if not p(x)`". Now the value of a negated assertion needs to be established before all facts of the negated predicate have been inferred. In particular,

> There may not be a unique T or F value for each assertion. For example, given only rule "`p(x) if not p(x)`", `p(a)` cannot be T because inferring it following the rule would require itself be F, and it cannot be F because it would lead to itself being T following the rule. That is, there may not be a 2-valued model.

In technical terms, the negation may be *non-stratified*. There are two best solutions to this that generalize a unique 2-valued model: a unique 3-valued model and a set of 2-valued models, as in well-founded semantics (WFS) and stable model semantics (SMS), respectively.

In a unique 3-valued model, when a unique T or F value cannot be established for an assertion, a third value, *undefined* (U), is used. For example, given only rule "`p(x) if not p(x)`", `p(a)` is U, in both WFS and founded semantics.

- With the semantics being 3-valued, when one cannot infer that an assertion is T, one should be able to express whether it is F or U when there is a choice. For example, given only rule "`p(x) if p(x)`", `p(a)` is not T, so `p(a)` may in general be F or U.
- WFS requires that such an assertion be F, even though common sense generally says that it is U. WFS attempts to be the same as in the case of 2-valued semantics, even though one is now in a 3-valued situation.
- Founded semantics supports both, allowing one to choose explicitly when there is a choice. Founded semantics is more expressive by supporting the choice. It is also more intuitive by supporting the common-sense choice for expressing ignorance.

For a set of 2-valued models, similar considerations motivate our constraint semantics. In particular, given only rule "`p(x) if not p(x)`", the semantics is the empty set, i.e., there is no model, in both SMS and constraint semantics, because no model can contain `p(a)` or `not p(a)`, for any `a`, because `p(a)` cannot be T or F as discussed above. However, given only rule "`p(x) if p(x)`", SMS requires that `p(a)` be F in all models, whereas constraint semantics allows the choice of `p(a)` being F in all models or being T in some models and F in other models.

Certain or uncertain. Founded semantics and constraint semantics first allow a predicate to be declared *certain* (i.e., each assertion of the predicate has one of two values: T, F) or *uncertain* (i.e., each assertion of the predicate has one of three values: T, F, U) when there is a choice. If a predicate is defined (as conclusions of rules) with use of non-stratified negation, then it must be declared uncertain, because it might not have a unique 2-valued model. Otherwise, it may be declared certain or uncertain.

- For a certain predicate, everything T must be given or inferred by following the rules, and the rest are F, in both founded semantics and constraint semantics.
- For an uncertain predicate, everything T or F must be given or inferred, and the rest are U in founded semantics. Constraint semantics then extends everything U to be combinations of T and F that satisfy all the rules and facts as constraints.

Complete or not. Founded semantics and constraint semantics then allow an uncertain predicate to be declared *complete*, i.e., all rules with that predicate in the conclusion are given.

- If a predicate is complete, then completion rules are added to define the negation of the predicate explicitly using the negation of the hypotheses of all given rules and facts of that predicates.
- Completion rules, if any, and given rules are used together to infer everything T and F. The rest are U in founded semantics, and are combinations of T and F in constraint semantics as described above.

Closed or not. Finally, founded semantics and constraint semantics can be extended to allow an uncertain, complete predicate to be declared *closed*, i.e., an assertion of the predicate is made F, called *self-false*, if inferring it to be T (respectively F) requires assuming itself to be T (respectively F).

- Determining self-false assertions is similar to determining unfounded sets in WFS. Repeatedly computing founded semantics and self-false assertions until a least fixed point is reached yields WFS.
- Among combinations of T and F values for assertions with U values in WFS, removing each combination that has self-false assertions that are not already F in that combination yields SMS.

Correspondence to prior semantics, more on motivation. Table 1 summarizes corresponding declarations that capture different assumptions under prior semantics; formal definitions and proofs for these and for additional relationships appear in the following sections. Note that founded semantics and constraint semantics allow additional combinations of declarations besides those in the table, because different predicates can have different declarations; founded semantics and constraint semantics are exponentially more expressive in this sense.

Some observations from Table 1 may help one better understand founded semantics and constraint semantics.

Prior Semantics	Kinds of Rules	New	Certain?	Complete?	Closed?	Theorem
Stratified	no non-stratified negation	Founded	yes	(implied yes)	(implied yes)	6
		Constraint				
First-Order Logic	any	Constraint	no	no	(implied no)	7
Fitting (Kripke-Kleene)	any	Founded	no*	yes	no	8
Supported		Constraint				12
WFS	any	Founded	no**	yes	yes	18
SMS		Constraint				19

Table 1: Correspondence between prior semantics and the new semantics, with declarations for all predicates, capturing different assumptions under prior semantics. * Some predicates can also be declared certain, per Theorem 3. ** Any predicate can also be declared certain if allowed, per Theorem 17.

- The 4 wide rows cover all combinations of allowed declarations (if all predicates have the same declarations).
- Wide row 1 is a special case of wide row 4, because being certain implies being complete and closed. So one could prefer to use only the latter two choices and omit the first choice. However, being certain is uniquely important, both for conceptual simplicity and practical efficiency:

  (1) It covers the vast class of database applications that do not use non-stratified negation, for which stratified semantics is universally accepted. It does not need to be understood by explicitly combining the latter two more sophisticated notions.

  (2) It allows founded semantics to match WFS for all example programs we found in the literature, with predicates being certain when possible and complete otherwise, but without the last, most sophisticated notion of being closed; and the semantics can be computed in linear time.
- Wide rows 2, 3, and 4 allow the assumptions about predicates that are uncertain, not complete, or not closed to be made explicitly.

In a sense, WFS uses F for both false and some kinds of ignorance (no knowledge of something must mean it is F), uses T for both true and some kinds of ignorance inferred through negation of F, and uses U for conflict, remaining kinds of ignorance from T and F, and imprecision; SMS resolves the ignorance in U, but not the ignorance in F and T. In contrast,

- founded semantics uses T only for true, F only for false, and U for conflict, ignorance, and imprecision;
- constraint semantics further differentiates among conflict, ignorance, and imprecision—corresponding to there being no model, multiple models, and a unique model, respectively, consistent with founded semantics.

After all, any easy-to-understand semantics must be consistent with the T and F assertions that can be inferred by exactly following the rules and completion rules starting from the given facts.

- Founded semantics is the maximum set of such T and F assertions, as a least fixed point of the given rules and completion rules if any, plus U for the remaining assertions.
- Constraint semantics is the set of combinations of all T and F assertions that are consistent with founded semantics and satisfy the rules as constraints.

Founded semantics without closed predicates can be computed easily and efficiently, as a least fixed point, contrasting with an alternating fixed point or iterated fixed point for computing WFS.

3 Language

We first consider Datalog with unrestricted negation in hypotheses. We extend it in Section 8 to allow unrestricted combinations of existential and universal quantifications and other features.

Datalog with unrestricted negation. A *program* in the core language is a finite set of rules of the following form, where any P_i may be preceded with $\neg$, and any P_i and Q over all rules may be declared certain or uncertain, and declared complete or not:

$$Q(X_1,\ldots,X_a) \leftarrow P_1(X_{11},\ldots,X_{1a_1}) \land \cdots \land P_h(X_{h1},\ldots,X_{ha_h}) \tag{1}$$

Symbols $\leftarrow$, $\land$, and $\neg$ indicate backward implication, conjunction, and negation, respectively; h is a natural number and is possibly 0, each P_i (respectively Q) is a predicate of finite number a_i (respectively a) of arguments, each X_{ij} and X_k is either a constant or a variable, and each variable in the arguments of Q must also be in the arguments of some P_i.

If $h = 0$, there is no P_i or X_{ij}, and each X_k must be a constant, in which case $Q(X_1,\ldots,X_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case each $P_i(X_{i1},\ldots,X_{ia_i})$ or $\neg P_i(X_{i1},\ldots,X_{ia_i})$ is called a *hypothesis* of the rule, and $Q(X_1,\ldots,X_a)$ is called the *conclusion* of the rule. The set of hypotheses of the rule is called the *body* of the rule.

A predicate declared certain means that each assertion of the predicate has a unique true (T) or false (F) value. A predicate declared uncertain means that each assertion of the predicate has a unique true, false, or undefined (U) value. A predicate declared complete means that all rules with that predicate in the conclusion are given in the program.

A predicate in the conclusion of a rule is said to be *defined* using the predicates or their negation in the hypotheses of the rule, and this defined-ness relation is transitive.

- A predicate must be declared uncertain if it is defined transitively using its own negation, or is defined using an uncertain predicate; otherwise, it may be declared certain or uncertain and is by default certain.
- A predicate may be declared complete or not only if it is uncertain, and it is by default complete.

In examples with no explicit specification of declarations, default declarations are used.

Rules of form (1) without negation are captured exactly by Datalog [CGT90, AHV95], a database query language based on the logic programming paradigm. Recursion in Datalog allows queries not expressible in relational algebra or relational calculus. Negation allows more sophisticated logic to be expressed directly. However, unrestricted negation in recursion has been the main challenge in defining the semantics of such a language, e.g., [AB94, Fit02], including whether the semantics should be 2-valued or 3-valued.

Example. We use `win`, the win-not-win game, as a running example, with default declarations: `move` is certain, and `win` is uncertain and complete. A move from position `x` to position `y` is represented by a fact `move(x,y)`. The following rule captures the win-not-win game: a position `x` is winning if there is a move from `x` to some position `y` and `y` is not winning. Arguments `x` and `y` are variables.

```
win(x) ← move(x,y) ∧ ¬ win(y)
```

Note that the declarations for predicates `move` and `win` are different. Other choices of declarations can lead to different results, e.g., see the first example in Section 9. ■

Notations. In arguments of predicates, we use letter sequences for variables, and use numbers and quoted strings for constants.

In presenting the semantics, in particular the completion rules, we use equality and the notations below for existential and universal quantifications, respectively, in the hypotheses of rules, and use negation in the conclusions.

$$\begin{array}{ll} \exists\ X_1, \ldots, X_n \mid Y & \text{existential quantification} \\ \forall\ X_1, \ldots, X_n \mid Y & \text{universal quantification} \end{array} \qquad (2)$$

The quantifications return T iff for some or all, respectively, combinations of values of $X_1, \ldots, X_n$, the value of Boolean expression Y is T. The domain of each quantified variable is the set of all constants in the program.

4 Formal definition of founded semantics and constraint semantics

Atoms, literals, consistency, and projection. Let π be a program. A predicate is *intensional* in π if it appears in the conclusion of at least one rule; otherwise, it is *extensional*.

An *atom* of π is a formula formed by applying a predicate symbol in π to constants in π. A *literal* of π is an atom of π or the negation of an atom of π. These are called *positive literals* and *negative literals*, respectively. The literals p and $\neg p$ are *complements* of each other. A set of literals is *consistent* if it does not contain a literal and its complement.

The *projection* of a program π onto a set S of predicates, denoted $\mathit{Proj}(\pi, S)$, contains all facts of π whose predicates are in S and all rules of π whose conclusions contain predicates in S.

Interpretations, ground instances, models, and derivability. An *interpretation* of π is a consistent set of literals of π. Interpretations are generally 3-valued: a literal p is *true* (T) in interpretation I if it is in I, is *false* (F) in I if its complement is in I, and is *undefined* (U) in I if neither it nor its complement is in I. An interpretation of π is *2-valued* if it contains, for each atom A of π, either A or its complement. An interpretation I is *2-valued for predicate* P if, for each atom A for P, I contains A or its complement. Interpretations are ordered by set inclusion $\subseteq$.

A *ground instance* of a rule R is any rule that can be obtained from R by expanding universal quantifications into conjunctions over all constants in the domain, and then instantiating the remaining variables with constants. For example, `q(a) ← p(a) ∧ r(b)` is a ground instance of `q(x) ← p(x) ∧ ∃ y | r(y)`.

An interpretation is a *model* of a program if it contains all facts in the program and satisfies all rules of the program, regarded as formulas in 3-valued logic [Fit85], i.e., for each ground instance of each rule, if the body is true, then so is the conclusion.

The *one-step derivability* operator T_π for program π performs one step of inference using rules of π, starting from a given interpretation. Formally, $C \in T_\pi(I)$ iff C is a fact of π or there is a ground instance R of a rule of π with conclusion C such that each hypothesis of R is true in interpretation I.

Dependency graph. The *dependency graph* $DG(\pi)$ of program π is a directed graph with a node for each predicate of π, and an edge from Q to P labeled $+$ (respectively, $-$) if a rule whose conclusion contains Q has a positive (respectively, negative) hypothesis that contains P. If the node for predicate P is in a cycle containing only positive edges, then P has *circular positive dependency* in π; if it is in a cycle containing a negative edge, then P has *circular negative dependency* in π.

4.1 Founded semantics

Intuitively, the *founded model* of a program π, denoted $\mathit{Founded}(\pi)$, is the least set of literals that are given as facts or can be inferred by repeated use of the rules. We define $\mathit{Founded}(\pi) = \mathit{UnNameNeg}(\mathit{LFPbySCC}(\mathit{NameNeg}(\mathit{Cmpl}(\pi))))$, where functions Cmpl, $\mathit{NameNeg}$, $\mathit{LFPbySCC}$, and $\mathit{UnNameNeg}$ are defined as follows.

Completion. The completion function, $\mathit{Cmpl}(\pi)$, returns the *completed program* of π. Formally, $\mathit{Cmpl}(\pi) = \mathit{AddInv}(\mathit{Combine}(\pi))$, where $\mathit{Combine}$ and AddInv are defined as follows.

The function $\mathit{Combine}(\pi)$ returns the program obtained from π by replacing the facts and rules defining each uncertain complete predicate Q with a single *combined rule* for Q, defined as follows. Transform the facts and rules defining Q so they all have the same conclusion $Q(V_1, \ldots, V_a)$, where $V_1, \ldots, V_a$ are fresh variables (i.e., not occurring in the given rules defining Q), by replacing each fact or rule $Q(X_1, \ldots, X_a) \leftarrow H_1 \wedge \cdots \wedge H_h$ with $Q(V_1, \ldots, V_a) \leftarrow (\exists\, Y_1, \ldots, Y_k \mid V_1 = X_1 \wedge \cdots \wedge V_a = X_a \wedge H_1 \wedge \cdots \wedge H_h)$, where $Y_1, \ldots, Y_k$ are all variables occurring in the given fact or rule. Combine the resulting rules for Q into a single rule defining Q whose body is the disjunction of the bodies of those rules. This combined rule for Q is logically equivalent to the original facts and rules for Q. Similar completion rules are used in Clark completion [Cla78] and Fitting semantics [Fit85].

Example. For the `win` example with default declarations, the rule for `win` becomes the following. For readability, we renamed variables to transform the equality conjuncts into tautologies and then eliminated them.

```
win(x) ← ∃ y | (move(x,y) ∧ ¬ win(y))
```

■

The function $\mathit{AddInv}(\pi)$ returns the program obtained from π by adding, for each uncertain complete predicate Q, a *completion rule* that derives negative literals for Q. The completion rule for Q is obtained from the inverse of the combined rule defining Q (recall that the inverse of $C \leftarrow B$ is $\neg C \leftarrow \neg B$), by putting the body of the rule in negation normal form, i.e., using laws of predicate logic to move negation inwards and eliminate double negations, so that negation is applied only to atoms.

Example. For the `win` example with default declarations, the added rule is

```
¬ win(x) ← ∀ y | (¬ move(x,y) ∨ win(y))
```

■

Least fixed point. The least fixed point is preceded and followed by functions that introduce and remove, respectively, new predicates representing the negations of the original predicates.

The function *NameNeg*(π) returns the program obtained from π by replacing each negative literal $\neg P(X_1, \ldots, X_a)$ with `n.`$P(X_1, \ldots, X_a)$, where the new predicate `n.`P represents the negation of predicate P.

Example. For the `win` example with default declarations, this yields:

```
win(x) ← ∃ y | (move(x,y) ∧ n.win(y))
n.win(x) ← ∀ y | (n.move(x,y) ∨ win(y))
```

■

The function *LFPbySCC*(π) uses a least fixed point to infer facts for each strongly connected component (SCC) in the dependency graph of π, as follows. Let $S_1, \ldots, S_n$ be a list of the SCCs in dependency order, so earlier SCCs do not depend on later ones; it is easy to show that any linearization of the dependency order leads to the same result for *LFPbySCC*. For convenience, we overload S to also denote the set of predicates in the SCC.

Define $\mathit{LFPbySCC}(\pi) = I_n$, where I_0 is the empty set and $I_i = \mathit{AddNeg}(\mathit{LFP}(T_{I_{i-1} \cup \mathit{Proj}(\pi, S_i)}), S_i)$ for $i \in 1..n$. *LFP* is the least fixed point operator. The least fixed point is well-defined, because the one-step derivability function $T_{I_{i-1} \cup \mathit{Proj}(\pi, S_i)}$ is monotonic, because the program π does not contain negation. The function $\mathit{AddNeg}(I, S)$ returns the interpretation obtained from interpretation I by adding *completion facts* for certain predicates in S to I; specifically, for each certain predicate P in S, for each combination of values $v_1, \ldots, v_a$ of arguments of P, if I does not contain $P(v_1, \ldots, v_a)$, then add `n.`$P(v_1, \ldots, v_a)$.

Example. For the `win` example with default declarations, the least fixed point calculation

1. infers `n.win(x)` for any `x` that does not have `move(x,y)` for any `y`, i.e., has no move to anywhere;

2. infers `win(x)` for any `x` that has `move(x,y)` for some `y` and `n.win(y)` has been inferred;

3. infers more `n.win(x)` for any `x` such that any `y` having `move(x,y)` has `win(y)`;

4. repeatedly does 2 and 3 above until a fixed point is reached.

■

The function *UnNameNeg*(I) returns the interpretation obtained from interpretation I by replacing each atom `n.`$P(X_1, \ldots, X_a)$ with $\neg P(X_1, \ldots, X_a)$.

Example. For the `win` example with default declarations, positions `x` for which `win(x)` is T, F, and U, respectively, in the founded model correspond exactly to the well-known winning, losing, and draw positions, respectively. In particular,

1. a losing position is one that either does not have a move to anywhere or has moves only to winning positions;

2. a winning position is one that has a move to a losing position; and

3. a draw position is one not satisfying either case above, i.e., it is in a cycle of moves that do not have a move to a losing position, called a *draw cycle*, or is a position that has only sequences of moves to positions in draw cycles.

■

4.2 Constraint semantics

Constraint semantics is a set of 2-valued models based on founded semantics. A *constraint model* of π is a consistent 2-valued interpretation M such that M is a model of $\mathit{Cmpl}(\pi)$ and $\mathit{Founded}(\pi) \subseteq M$. We define $\mathit{Constraint}(\pi)$ to be the set of constraint models of π. Constraint models can be computed from $\mathit{Founded}(\pi)$ by iterating over all assignments of true and false to atoms that are undefined in $\mathit{Founded}(\pi)$, and checking which of the resulting interpretations satisfy all rules in $\mathit{Cmpl}(\pi)$.

Example. For the `win` example with default declarations, draw positions (i.e., positions for which `win` is undefined) are in draw cycles, i.e., cycles that do not have a `move` to a `n.win` position, or are positions that have only a sequence of moves to positions in draw cycles.

1. If some SCC has draw cycles of only odd lengths, then there is no satisfying assignment of T and F to `win` for positions in the SCC, so there are no constraint models of the program.

2. If some SCC has draw cycles of only even lengths, then there are two satisfying assignments of T and F to `win` for positions in the SCC, with the truth values alternating between T and F around each cycle, and with the second truth assignment obtained from the first by swapping T and F. The total number of constraint models of such SCCs is exponential in the number of such SCCs.

■

5 Properties of founded semantics and constraint semantics

Proofs of theorems appear in Appendix A.

Consistency and correctness. The most important properties are consistency and correctness.
Theorem 1. The founded model and constraint models of a program π are consistent.
Theorem 2. The founded model of a program π is a model of π and $\mathit{Cmpl}(\pi)$. The constraint models of π are 2-valued models of π and $\mathit{Cmpl}(\pi)$.

Equivalent declarations. When a predicate can be declared certain, other declarations sometimes have the same effect.
Theorem 3. Let P be a predicate that can be declared certain in program π. Let S be the set containing P and the predicates on which P depends. If each predicate in S does not have positive circular dependency in π, then the founded semantics of π is the same if predicates in S are certain or are uncertain and complete.

Same SCC, same certainty. All predicates in an SCC have the same certainty.
Theorem 4. For every program, for every SCC S in its dependence graph, all predicates in S are certain, or all of them are uncertain.

Higher-order programming. Higher-order logic programs, in languages such as HiLog, can be encoded as first-order logic programs by a semantics-preserving transformation that replaces uses of the original predicates with uses of a single predicate `holds` whose first argument is the name of an original predicate [CKW93]. For example, `win(x)` is replaced with `holds(win,x)`. This transformation merges a set of predicates into a single predicate, facilitating higher-order programming.

We show that founded semantics and constraint semantics are preserved by merging of *compatible* predicates, defined below, if a simple type system is used to distinguish the constants in the original program from the new constants representing the original predicates.

We extend the language with a simple type system. A type denotes a set of constants. Each predicate has a type signature that specifies the type of each argument. A program is well-typed if, in each rule or fact, (1) each constant belongs to the type of the argument where the constant occurs, and (2) for each variable, all its occurrences are as arguments with the same type. In the semantics, the values of predicate arguments are restricted to the appropriate type.

Predicates of program π are *compatible* if they are in the same SCC in $DG(\pi)$ and have the same arity, same type signature, and (if uncertain) same completeness declaration. For a set S of compatible predicates of program π with arity a and type signature $T_1, \ldots, T_a$, the *predicate-merge transformation* $Merge_S$ transforms π into a program $Merge_S(\pi)$ in which predicates in S are replaced with a single fresh predicate `holds` whose first parameter ranges over S, and which has the same completeness declaration as the predicates in S. Each atom A in a rule or fact of π is replaced with $MergeAtom_S(A)$, where the function $MergeAtom_S$ on atoms is defined by: $MergeAtom_S(P(X_1, \ldots, X_a))$ equals `holds('`P`', `X_1`, ..., `X_a`)` if $P \in S$ and equals $P(X_1, \ldots, X_a)$ otherwise. We extend $MergeAtom_S$ pointwise to a function on sets of atoms, used for properties of founded semantics, and a function on sets of sets of atoms, used for properties of constraint semantics. The predicate-merge transformation introduces S as a new type. The type signature of `holds` is $S, T_1, \ldots, T_a$.
Theorem 5. Let S be a set of compatible predicates of program π. Then $Merge_S(\pi)$ and π have the same founded semantics, in the sense that $Founded(Merge_S(\pi)) = MergeAtom_S(Founded(\pi))$. $Merge_S(\pi)$ and π also have the same constraint semantics, in the sense that $Constraint(Merge_S(\pi))$ $= MergeAtom_S(Constraint(\pi))$.

6 Comparison with other semantics

Stratified semantics. A program π has *stratified negation* if it does not contain predicates with circular negative dependencies. Such a program has a well-known and universally accepted semantics that defines a unique 2-valued model, denoted $Stratified(\pi)$, as discussed in Section 2.
Theorem 6. For a program π with stratified negation and in which all predicates are certain, $Founded(\pi) = Stratified(\pi)$.

First-order logic. The next theorem relates constraint models with the meaning of a program regarded as a set of formulas in first-order logic; recall that the definition of a model of a program also regards rules as logical formulas.
Theorem 7. For a program π in which all predicates are uncertain and not complete, the constraint models of π are exactly the 2-valued models of π.

Fitting semantics. Fitting [Fit85] defines an interpretation to be a model of a program iff it satisfies a formula that we denote as $CCmpl(\pi)$. This formula is Fitting's 3-valued-logic version of the Clark completion of π [Cla78]. Briefly, $CCmpl(\pi) = CCmpl_D(\pi) \wedge CCmpl_U(\pi)$, where $CCmpl_D(\pi)$ is the conjunction of formulas corresponding to the combined rules introduced by *Combine* except with $\leftarrow$ replaced with $\cong$ (which is called "complete equivalence" and means "same

truth value"), and $\mathit{CCmpl}_U(\pi)$ is the conjunction of formulas stating that predicates not used in any fact or the conclusion of any rule are false for all arguments. The *Fitting model* of a program π, denoted $\mathit{Fitting}(\pi)$, is the least model of $\mathit{CCmpl}(\pi)$ [Fit85].
Theorem 8. For a program π in which all predicates are uncertain and complete, $\mathit{Founded}(\pi) = \mathit{Fitting}(\pi)$.
Theorem 3 implies that Theorem 8 also holds if predicates that do not have positive circular dependency and can be declared certain are declared certain instead of uncertain and complete.

Founded semantics for some declarations is less defined than or equal to Fitting semantics, as stated in the following theorem.
Theorem 9. (a) For a program π in which all intensional predicates are uncertain and complete, $\mathit{Founded}(\pi) \subseteq \mathit{Fitting}(\pi)$. (b) If, furthermore, some extensional predicate is uncertain, and some positive literal p for some uncertain extensional predicate does not appear in π, then $\mathit{Founded}(\pi) \subset \mathit{Fitting}(\pi)$.
A simple program π_6 for which the inclusion in Theorem 9 is strict, as in part (b) of the theorem, is program 6 in Table 2, which has only one rule $\mathtt{q} \leftarrow \mathtt{p}$. With both predicates being uncertain and complete, $\mathit{Founded}(\pi_6) = \emptyset$ and $\mathit{Fitting}(\pi_6) = \{\neg\mathtt{p}, \neg\mathtt{q}\}$.

Founded semantics for default declarations is at least as defined as Fitting semantics, as stated in the following theorem.
Theorem 10. (a) For a program π in which all predicates have default declarations as certain or uncertain and complete, $\mathit{Fitting}(\pi) \subseteq \mathit{Founded}(\pi)$. (b) If, furthermore, $\mathit{Fitting}(\pi)$ is not 2-valued for some certain intensional predicate P, then $\mathit{Fitting}(\pi) \subset \mathit{Founded}(\pi)$.
A simple program π_3 for which the inclusion in Theorem 10 is strict, as in part (b) of the theorem, is program 3 in Table 2, which has only one rule $\mathtt{q} \leftarrow \mathtt{q}$. $\mathit{Fitting}(\pi_3) = \emptyset$ and $\mathit{Founded}(\pi_3) = \{\neg\,\mathtt{q}\}$.

Well-founded semantics. The *well-founded model* of a program π, denoted $\mathit{WFS}(\pi)$, is the least fixed point of a monotone operator W_π on interpretations, defined as follows [VRS91]. A set U_0 of atoms of a program π is an *unfounded set* of π with respect to an interpretation I of π iff, for each atom A in U_0, for each ground instance R of a rule of π with conclusion A, either (1) some hypothesis of R is false in I or (2) some positive hypothesis of R is in U_0. Intuitively, the atoms in U_0 can be set to false, because each rule R whose conclusion is in U_0 either has a hypothesis already known to be false or has a hypothesis in U_0 (which will be set to false). Let $U_\pi(I)$ be the *greatest unfounded set* of program π with respect to interpretation I. For a set S of atoms, let $\neg\cdot S$ denote the set containing the negations of those atoms. W_π is defined by $W_\pi(I) = T_\pi(I) \cup \neg\cdot U_\pi(I)$. The well-founded model $\mathit{WFS}(\pi)$ is a model of $\mathit{CCmpl}(\pi)$, so $\mathit{Fitting}(\pi) \subseteq \mathit{WFS}(\pi)$ for all programs π [VRS91].
Theorem 11. For every program π, $\mathit{Founded}(\pi) \subseteq \mathit{WFS}(\pi)$.
The inclusion in Theorem 11 is strict for program 8 in Table 2, denoted π_8, which has only one rule $\mathtt{q} \leftarrow \neg\,\mathtt{q} \wedge \mathtt{q}$. $\mathit{Founded}(\pi_8) = \emptyset$ and $\mathit{WFS}(\pi_8) = \{\neg\mathtt{q}\}$.

Supported models. Supported model semantics of a logic program is a set of 2-valued models. An interpretation I is a *supported model* of π if I is 2-valued and I is a fixed point of the one-step derivability operator T_π [ABW88]. Let $\mathit{Supported}(\pi)$ denote the set of supported models of π. Supported models, unlike Fitting semantics and WFS, allow atoms to be set to true when they have circular positive dependency and nothing else, like the atom $\mathtt{q}$ in program π_3 described above.

The following three theorems relating constraint semantics with supported model semantics are analogous to the three theorems relating founded semantics with Fitting semantics.
Theorem 12. For a program π in which all predicates are uncertain and complete, $\mathit{Supported}(\pi) = \mathit{Constraint}(\pi)$.
Theorem 3 implies that Theorem 12 also holds if predicates that do not have positive circular dependency and can be declared certain are declared certain instead of uncertain and complete.
Theorem 13. For a program π in which all intensional predicates are uncertain and complete, $\mathit{Supported}(\pi) \subseteq \mathit{Constraint}(\pi)$.
The inclusion in Theorem 13 is strict for the program π_6 described above. $\mathit{Supported}(\pi_6) = \{\{\neg\,\mathtt{p}, \neg\,\mathtt{q}\}\}$ and $\mathit{Constraint}(\pi_6) = \{\{\mathtt{p}, \mathtt{q}\}, \{\neg\,\mathtt{p}, \neg\,\mathtt{q}\}\}$.
Theorem 14. For a program π in which all predicates have default declarations as certain or uncertain and complete, $\mathit{Constraint}(\pi) \subseteq \mathit{Supported}(\pi)$.
The inclusion in Theorem 14 is strict for the program π_3 described above. $\mathit{Constraint}(\pi_3) = \{\{\neg\,\mathtt{q}\}\}$ and $\mathit{Supported}(\pi_6) = \{\{\mathtt{q}\}, \{\neg\,\mathtt{q}\}\}$.

Stable models. Gelfond and Lifschitz define *stable model semantics* (SMS) of logic programs [GL88]. They define the *stable models* of a program π to be the 2-valued interpretations of π that are fixed points of a particular transformation. Van Gelder et al. proved that the stable models of π are exactly the 2-valued fixed points of the operator W_π described above [VRS91, Theorem 5.4]. Let $\mathit{SMS}(\pi)$ denote the set of stable models of π.
Theorem 15. For every program π, $\mathit{SMS}(\pi) \subseteq \mathit{Constraint}(\pi)$.
The inclusion in Theorem 15 is strict for program 7 in Table 2, denoted π_7, which has two rules $\mathtt{q} \leftarrow \neg\,\mathtt{q}$ and $\mathtt{q} \leftarrow \mathtt{q}$. $\mathit{SMS}(\pi_7) = \emptyset$ and $\mathit{Constraint}(\pi_7) = \{\mathtt{q}\}$.

7 Comparison of semantics for well-known small examples and more

Table 2 shows well-known example rules and more for tricky boundary cases in the semantics, where all uncertain predicates that are in a conclusion are declared complete, but not closed, and shows different semantics for them.

- Programs 1 and 2 contain only negative cycles. All three of Founded, WFS, and Fitting agree. All three of Constraint, SMS, and Supported agree.
- Programs 3 and 4 contain only positive cycles. Founded for certain agrees with WFS; Founded for uncertain agrees with Fitting. Constraint for certain agrees with SMS; Constraint for uncertain agrees with Supported.
- Programs 5 and 6 contain no cycles. Founded for certain agrees with WFS and Fitting; Founded for uncertain has more undefined. Constraint for certain agrees with SMS and Supported; Constraint for uncertain has more models.
- Programs 7 and 8 contain both negative and positive cycles. For program 7 where $\neg\,\mathtt{q}$ and $\mathtt{q}$ are disjunctive, all three of Founded, WFS, and Fitting agree; Constraint and Supported agree, but SMS has no model. For program 8 where $\neg\,\mathtt{q}$ and $\mathtt{q}$ are conjunctive, Founded and Fitting agree, but WFS has $\mathtt{q}$ being F; all three of Constraint, SMS, and Supported agree.

For all 8 programs, with default complete but not closed predicates, we have the following:

	Program	Founded (not closed)		WFS	Fitting (Kripke -Kleene)	Constraint (not closed)		SMS	Supported
		uncertain	certain			uncertain	certain		
1	$q \leftarrow \neg q$	$\{\underline{q}\}$	–	$\{\underline{q}\}$	$\{\underline{q}\}$	no model	–	no model	no model
2	$q \leftarrow \neg p$ $p \leftarrow \neg q$	$\{\underline{p}, \underline{q}\}$	–	$\{\underline{p}, \underline{q}\}$	$\{\underline{p}, \underline{q}\}$	$\{p, \overline{q}\}$,$\{\overline{p}, q\}$	–	$\{p, \overline{q}\}$,$\{\overline{p}, q\}$	$\{p, \overline{q}\}$,$\{\overline{p}, q\}$
3	$q \leftarrow q$	$\{\underline{q}\}$	$\{\overline{q}\}$	$\{\overline{q}\}$	$\{\underline{q}\}$	$\{q\}$,$\{\overline{q}\}$	$\{\overline{q}\}$	$\{\overline{q}\}$	$\{q\}$,$\{\overline{q}\}$
4	$q \leftarrow p$ $p \leftarrow q$	$\{\underline{p}, \underline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\underline{p}, \underline{q}\}$	$\{p, q\}$,$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{p, q\}$,$\{\overline{p}, \overline{q}\}$
5	$q \leftarrow \neg p$	$\{\underline{p}, \underline{q}\}$	$\{\overline{p}, q\}$	$\{\overline{p}, q\}$	$\{\overline{p}, q\}$	$\{p, \overline{q}\}$,$\{\overline{p}, q\}$	$\{\overline{p}, q\}$	$\{\overline{p}, q\}$	$\{\overline{p}, q\}$
6	$q \leftarrow p$	$\{\underline{p}, \underline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{p, q\}$,$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$	$\{\overline{p}, \overline{q}\}$
7	$q \leftarrow \neg q$ $q \leftarrow q$	$\{\underline{q}\}$	–	$\{\underline{q}\}$	$\{\underline{q}\}$	$\{q\}$	–	no model	$\{q\}$
8	$q \leftarrow \neg q$ $\wedge\ q$	$\{\underline{q}\}$	–	$\{\overline{q}\}$	$\{\underline{q}\}$	$\{\overline{q}\}$	–	$\{\overline{q}\}$	$\{\overline{q}\}$

Table 2: Different semantics for programs where all uncertain predicates that are in a conclusion are declared complete, but not closed. "uncertain" means all predicates in the program are declared uncertain. "certain" means all predicates in the program that can be declared certain are declared certain; "–" means no predicates can be declared certain, so the semantics is the same as "uncertain". p, $\overline{p}$ and $\underline{p}$ mean p is T, F, and U, respectively.

- If all predicates are the default certain or uncertain, then Founded agrees with WFS, and Constraint agrees with SMS, with one exception for each:

  (1) Program 7 concludes `q` whether `q` is F or T, so SMS having no model is an extreme outlier among all 6 semantics and is not consistent with common sense.

  (2) Program 8 concludes `q` if `q` is F and T, so Founded semantics with `q` being U is imprecise, but Constraint has `q` being F. WFS has `q` being F because it uses F for ignorance.

- If predicates not in any conclusion are certain (not shown in Table 2 but only needed for `p` in programs 5 and 6), and other predicates are uncertain, then Founded equals Fitting, and Constraint equals Supported, as captured in Theorems 8 and 12, respectively.

- If all predicates are uncertain, then Founded has all values being U, capturing the well-known unclear situations in all these programs, and Constraint gives all different models except for programs 2 and 5, and programs 4 and 6, which are pair-wise equivalent under completion, capturing exactly the differences among all these programs.

Finally, if all predicates in these programs are not complete, then Founded and Constraint are the same as in Table 2 except that Constraint for uncertain becomes equivalent to truth values in first-order logic: programs 1 and 8 have an additional model, $\{q\}$, program 6 has an additional model, $\{\overline{p}, q\}$, and programs 2 and 5 have an additional model, $\{p,q\}$.

8 Computational complexity and extensions

Computing founded semantics and constraint semantics.
***Theorem 16*.** Computing founded semantics is linear time in the size of the ground program.

Proof. First ground all given rules, using any grounding. Then add completion rules, if any, by adding an inverse rule for each group of the grounded given rules that have the same conclusion, yielding ground completion rules of the same asymptotic size as the grounded given rules.

Now compute the least fixed point for each SCC of the resulting ground rules using a previous method [LS09]. To do so, first introduce a new intermediate predicate and rule for each conjunction and disjunction in the rules, yielding a new set of rules of the same asymptotic size. In computing the least fixed point, each resulting rule incurs at most one rule firing because there are no variables in the rule, and each firing takes worst-case $O(1)$ time. Thus, the total time is worst-case linear in the size of all ground rules and therefore in the size of the grounded given rules. ■

The size of the ground program is polynomial in the size n of input data, i.e., the given facts, because each variable in each rule can be instantiated at most $O(n)$ times (because the domain size is at most n), and there is a fixed number of variables in each rule, and a fixed size of the given rules. Precisely, the size of the ground program is in the worst case $O(n^k \times r)$, where k is the maximum number of variables in a rule, and r is the size of the given rules.

Computing constraint semantics may take exponential time in the size of the input data, because in the worst case, all assertions of all predicates may have U values in founded semantics, and there is an exponential number of combinations of T and F values of all assertions, where each combination may be checked for whether it satisfies the constraints imposed by all rules.

These complexity analyses also apply to the extensions below except that computing founded semantics with closed predicates may take quadratic time in the size of the ground program, because of repeated computation of founded semantics and self-false assertions.

Closed predicate assumption. We can extend the language to support declaration of uncertain complete predicates as *closed*. Informally, this means that an atom A of the predicate is false in an interpretation I, called *self-false* in I, if every ground instance of rules that concludes A, or recursively concludes some hypothesis of that rule instance, has a hypothesis that is false or, recursively, is self-false in I. Self-false atoms are elements of unfounded sets.

Formally, $\mathit{SelfFalse}_\pi(I)$, the set of self-false atoms of program π with respect to interpretation I, is defined in the same way as the greatest unfounded set of π with respect to I, except replacing "some positive hypothesis of R is in U_0" with "some positive hypothesis of R for a closed predicate is in U_0". The founded semantics of this extended language is defined by repeatedly computing the semantics as per Section 4 and then setting self-false atoms to false, until a least fixed point is reached. Formally, the founded semantics is $\mathit{FoundedClosed}(\pi) = \mathit{LFP}(F_\pi)$, where $F_\pi(I) = \mathit{Founded}(\pi \cup I) \cup \neg \cdot \mathit{SelfFalse}_\pi(\mathit{Founded}(\pi \cup I))$.

The constraint semantics for this extended language includes only interpretations that contain the negative literals required by the closed declarations. Formally, a *constraint model* of a program π with closed declarations is a consistent 2-valued interpretation M such that M is a model of $\mathit{Cmpl}(\pi)$, $\mathit{FoundedClosed}(\pi) \subseteq M$, and $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq M$. Let $\mathit{ConstraintClosed}(\pi)$ denote the set of constraint models of π.

The next theorem states that changing predicate declarations from uncertain, complete, and closed to certain when allowed, or vice versa, preserves founded and constraint semantics. Theorem 4 implies that this change needs to be made for all predicates in an SCC.

Theorem 17. Let π be a program. Let S be an SCC in its dependence graph containing only predicates that are uncertain, complete, and closed. Let π' be a program identical to π except that all predicates in S are declared certain. Note that, for the declarations in both programs to be allowed, predicates in all SCCs that follow S in dependency order must be uncertain, predicates in all SCCs that precede S in dependency order must be certain, and predicates in S must not have circular negative dependency. Then $\mathit{FoundedClosed}(\pi) = \mathit{FoundedClosed}(\pi')$ and $\mathit{ConstraintClosed}(\pi) = \mathit{ConstraintClosed}(\pi')$.

***Theorem 18*.** For a program π in which every predicate is uncertain, complete, and closed, $\mathit{FoundedClosed}(\pi) = \mathit{WFS}(\pi)$.

***Theorem 19*.** For a program π in which every predicate is uncertain, complete, and closed, $\mathit{ConstraintClosed}(\pi) = \mathit{SMS}(\pi)$.

Note, however, that founded semantics for default declarations (certain when possible and complete otherwise) allows the number of repetitions for computing self-false atoms to be greatly reduced, even to zero, compared with WFS that does repeated computation of unfounded sets.

In all examples we have found in the literature, and all natural examples we have been able to think of, founded semantics for default declarations, without closed predicate assumption, infers the same result as WFS. However, while founded semantics computes a single least fixed point without the outer repetition and is worst-case linear time, WFS computes an alternating fixed point or iterated fixed point and is worst-case quadratic. In fact, we have not found any natural example showing that an actual quadratic-time alternating or iterated fixed-point for computing WFS is needed.[1]

Unrestricted existential and universal quantifications in hypotheses. We extend the language to allow unrestricted combinations of existential and universal quantifications as well as negation, conjunction, and disjunction in hypotheses. The domain of each quantified variable is the set of all constants in the program.

***Example*.** For the `win` example, the following two rules may be given instead:

```
win(x) ← ∃ y | move(x,y) ∧ lose(y)
lose(x) ← ∀ y | ¬ move(x,y) ∨ win(y)
```

■

The semantics in Section 4 is easily extended to accommodate this extension: these constructs simply need to be interpreted, using their 3-valued logic semantics [Fit85], when defining one-step derivability. Theorems 1–5 hold for this extended language. The other semantics discussed in Section 6 are not defined for this extension, thus we do not have theorems relating to them.

Negation in facts and conclusions. We extend the language to allow negation in given facts and in conclusions of given rules; such facts and rules are said to be *negative*. The Yale shooting example in Section 9 is a simple example.

The definition of founded semantics applies directly to this extension, because it already introduces and handles negative rules, and it already infers and handles negative facts. Note that *Combine* combines only positive facts and positive rules to form combined rules; negative facts and negative rules are copied unchanged into the completed program.

With this extension, a program and hence its founded model may be inconsistent; for example, a program could contain or imply `p` and `¬p`. Thus, Theorem 1 does not hold for such programs. When

[1]Even a contrived example that demonstrates the worst-case quadratic-time computation of WFS has been challenging to find. For example, the quadratic-time example in [Zuk01] turns out to be linear in XSB; after significant effort between us and Warren, we found a much more sophisticated example that appears to take quadratic time, but a remaining bug in XSB makes the correctness of its computation unclear.

the founded model is inconsistent, the inconsistent literals in it can easily be reported to the user. When the founded model is consistent, the definition of constraint semantics applies directly, and Theorems 2–5 hold. The other semantics discussed in Section 6 are not defined for this extended language, so we do not have theorems relating to them.

9 Additional examples

We discuss the semantics of several main well-known examples.

Win-not-win game. This is the running example with the `move` facts and a single rule that defines `win` recursively using its own negation.

With the default declaration that `move` is certain and `win` is uncertain and complete, founded semantics and constraint semantics are as discussed in Section 4. There is no circular positive dependency. Fitting semantics and WFS are the same as founded semantics, and supported model semantics and SMS are the same as constraint semantics.

If `move` is declared uncertain instead of the default of being certain, then moves not in the given `move` facts have U values, not allowing any `n.win` or `win` facts to be inferred. Therefore, founded semantics infers `move` to be T for the given `move` facts and U for all other pairs of positions, and `win` to be U for all positions. Constraint semantics extends the T values for `move` with all combinations of T and F values for the U values of `move` and `win` such that the combination satisfies the given rule and completion rule.

Graph reachability. A source vertex `x` is represented by a fact `source(x)`. An edge from a vertex `x` to a vertex `y` is represented by a fact `edge(x,y)`. The following two rules capture graph reachability, i.e., the set of vertices reachable from source vertices by following edges.

```
reach(x) ← source(x)
reach(y) ← edge(x,y) ∧ reach(x)
```

In the dependency graph, each predicate is in a separate SCC, and the SCC for `reach` is ordered after the other two. There is no negation in this program.

With the default declarations of all predicates being certain, no completion rules are added. The least fixed point computation for founded semantics infers `reach` to be T for all vertices that are source vertices or are reachable from source vertices by following edges, as desired. For the remaining vertices, `reach` is F. Constraint semantics is the same. These are the same as WFS and SMS.

If `reach` is declared uncertain and complete, but not closed, then after completion, we obtain

```
reach(x) ← source(x) ∨ (∃ y | (edge(y,x) ∧ reach(y)))
n.reach(x) ← n.source(x) ∧ (∀ y | (n.edge(y,x) ∨ n.reach(y)))
```

The least fixed point computation for founded semantics infers `reach` to be T for all reachable vertices as when predicates are certain, and infers `reach` to be F for all vertices that are not source vertices and that have no in-coming edge at all or have in-coming edges only from vertices for which `reach` is F. For the remaining vertices, i.e., those that are not reachable from the source vertices but are in cycles of edges, `reach` is U. Constraint semantics extends this model with all combinations of T and F values for the U values such that the combination satisfies the given rules and completion rule. These are the same as in Fitting semantics and supported model semantics, respectively.

Russell's paradox. Russell's paradox is well known as the barber paradox. The barber is a man who shaves all those men, and those men only, who do not shave themselves, specified using the following fact and rule:

```
man('barber')
shave('barber',x) ← man(x) ∧ ¬ shave(x,x)
```

The question is: Does the barber shave himself? That is: What is the value of `shave('barber', 'barber')`?

With the default declarations that `man` is certain, and `shave` is uncertain (because `shave` is defined recursively using its own negation) and complete, which captures "those men only", the completion step adds the rule

```
¬ shave(y,x) ← y≠'barbar' ∨ ¬ man(x) ∨ shave(x,x)
```

The completed program, after eliminating negation, is

```
shave('barber',x) ← man(x) ∧ n.shave(x,x)
man('barber')
n.shave(y,x) ← y≠'barbar' ∨ n.man(x) ∨ shave(x,x))
```

The least fixed point computation for founded semantics infers no T or F facts of `shave`, leaving `shave('barber','barber')` to be U. Constraint semantics has no model. These results correspond to WFS and SMS, respectively. Our results show the exact assumptions (certain and complete) and desired outcome restrictions (allowing U values or allowing only T and F values) that lead to different solutions.

If there are other men besides the barber, then founded semantics also infers `shave('barber',x)` for all man `x` except `'barber'` to be T, and `shave(y,x)` for all man `y` except `'barber'` and for all man `x` to be F, leaving only `shave('barber','barber')` to be U. For example, if there is also a fact `man('tom')`, then founded semantics also infers `shave('barber','tom')` to be T, and `shave('tom','barber')` and `shave('tom','tom')` to be F. Constraint semantics has no model. These results again correspond to WFS and SMS, respectively, and show the exact assumptions and outcome restrictions that lead to different solutions.

Even numbers. In this example, even numbers are defined by the predicate `even`, and natural numbers in order are given using the predicate `succ`.

```
even(n) ← succ(m,n) ∧ ¬ even(m)
even(0)
succ(0,1)
succ(1,2)
succ(2,3)
```

With the default declarations that `succ` is certain and `even` is uncertain and complete, the following completion rule is added:

```
n.even(n) ← n≠0 ∧ (∀ m | n.succ(m,n) ∨ even(m))
```

Founded semantics infers that `even(1)` is F, `even(2)` is T, and `even(3)` is F. Constraint semantics is the same. These results are the same as WFS and SMS.

Yale shooting. This example is about whether a turkey is alive, given some facts and rules about whether and when a gun is loaded, specified below. It uses the extension that allows negative facts and negative conclusions.

```
alive(0)
¬ loaded(0)
loaded(1)
¬ alive(3) ← loaded(2)
```

Assume both predicates `alive` and `loaded` are declared uncertain and not complete. In the dependency graph, there are two SCCs: one with `loaded`, one with `alive`, and the former is ordered before the latter. Founded semantics infers that `loaded(0)` is F, `loaded(1)` is T, `loaded(2)` and `loaded(3)` are U, `alive(0)` is T, and `alive(1)`, `alive(2)`, and `alive(3)` are U. Constraint semantics has multiple models, some containing that `loaded(2)` is T and `alive(3)` is F, and some containing that `loaded(2)` is F and `alive(3)` is T. These confirm the well-known outcomes.

If there are other facts and rules that give or infer `loaded(2)` to be T, then `alive(3)` is F in both founded and constraint semantics. This is again the well-known outcome.

Variant of Yale shooting. This is a variant of the Yale shooting problem, copied from [VRS91]:

```
noise(T) ← loaded(T) ∧ shoots(T).
loaded(0).
loaded(T) ← succ(S,T) ∧ loaded(S) ∧ ¬ shoots(S).
shoots(T) ← triggers(T).
triggers(1).
succ(0,1).
```

There is no circular negative dependency, so the default is that all predicates are certain. In this case, no completion rules are added. Founded semantics and constraint semantics both yield that `succ(0,1)` and `loaded(0)` are T; `trigger(0)`, `shoots(0)`, and `noise(0)` are F; and `loaded(1)`, `trigger(1)`, `shoots(1)`, and `noise(1)` are T. This is the same as WFS, Fitting semantics, SMS, and supported models.

10 Related work and conclusion

There is a large literature on logic language semantics and implementations. Several overview articles [AB94, Prz94, RU95, Fit02] give a good sense of the challenges when there is unrestricted negation. We discuss major prior semantics here; major implementations are as discussed in Section 1.

Clark [Cla78] describes completion of logic programs to give a semantics for negation as failure. Numerous others, e.g., [LT84, ST84, JLM86, Cha88, FRTW88, Stu91], describe similar additions. Fitting [Fit85] presents a semantics, called Fitting semantics or Kripke-Kleene semantics, that aims to give a least 3-valued model. Apt et al. [ABW88] defines supported model semantics, which is a set of 2-valued models; the models correspond to extensions of the Fitting model. Apt et al. [ABW88] introduces stratified semantics. WFS [VRS91] also gives a 3-valued model but aims to maximize false values. SMS [GL88] also gives a set of 2-valued models and aims to maximize false values. Other formalisms and semantics include partial stable models, also called stationary models [Prz94], and FO(ID), for first-order logic with inductive definitions [DT08]. There are also many studies that relate different semantics, e.g., [Dun92, LZ04].

Our founded semantics, which extends to constraint semantics, is unique in that it allows predicates to be specified as certain or uncertain, as complete or not, and as closed or not. These choices clearly and explicitly capture the different assumptions one can have about the predicates, rules, and reasoning, and capture them as simple and precise binary choices, unlike the well-known closed-world assumption and open-world assumption, and allow different combinations of assumptions to co-exist naturally. These choices make founded and constraint semantics more expressive and intuitive. Instead of using many separate semantics, one just need to make the assumptions explicit; the same underlying logic is used for inference. In this way, founded semantics and constraint semantics unify different semantics.

In addition, founded semantics and constraint semantics are completely declarative, as a least fixed point and as constraint satisfaction, respectively. Our default declarations without closed predicates lead to the same semantics as WFS and SMS for all natural examples we have found. Additionally, founded semantics without closed predicates can be computed in linear time in the size of the ground program, as opposed to quadratic time for WFS.

Liu and Stoller [LS20] creates a unified language for design and analysis, DA logic, based on founded semantics and constraint semantics, to support the power and ease of programming with different intended semantics. It provides meta-constraints to specify different assumptions, supports the use of uncertain information in the form of either undefined values or possible combinations of values, and, for composability, introduces knowledge units that can be instantiated by any new predicates, including predicates with additional arguments.

There are many directions for future study, including additional relationships with prior semantics, further extensions, efficient implementations, and applications.

Acknowledgment. We thank David S. Warren, Michael Kifer, Anil Nerode, Tuncay Tekle, Molham Aref, Marc Denecker, Cordell Green, Goyal Gupta, John Hooker, Bob Kowalski, Fangzhen Lin, Zuoquan Lin, Alberto Pettorossi, Maurizio Proietti, Carlo Zaniolo, Neng-Fa Zhou, and many others for helpful comments and discussions on logic languages, semantics, and efficient computations.

References

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19:9–71, 1994.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufman, 1988.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level.* Addison-Wesley, 1995.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Springer, 1990.

[Cha88] David Chan. Constructive negation based on the completed database. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 111–125. MIT Press, 1988.

[CKW93] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[Cla78] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[DT08] Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2):14, 2008.

[Dun92] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 105(1):7–25, 1992.

[Fit85] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Fit02] Melvin Fitting. Fixpoint semantics for logic programming: A survey. *Theoretical Computer Science*, 278(1):25–51, 2002.

[FRTW88] Norman Y. Foo, Anand S. Rao, Andrew Taylor, and Adrian Walker. Deduced relevant types and constructive negation. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 126–139, 1988.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[JLM86] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. Some issues and trends in the semantics of logic programming. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 223–241. Springer, 1986.

[LS09] Yanhong A. Liu and Scott D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):1–38, 2009.

[LS18] Yanhong A. Liu and Scott D. Stoller. Founded semantics and constraint semantics of logic rules. In *Proceedings of the International Symposium on Logical Foundations of Computer Science*, pages 221–241. Springer, 2018.

[LS20] Yanhong A. Liu and Scott D. Stoller. Knowledge of uncertain worlds: Programming with logical constraints. In *Proceedings of the International Symposium on Logical Foundations of Computer Science*, pages 111–127. Springer, 2020.

[LT84] John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[LZ04] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[Prz94] Teodor C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(3):141–187, 1994.

[RU95] Raghu Ramakrishnan and Jeffrey D Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[ST84] Taisuke Sato and Hisao Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 195–201, 1984.

[Stu91] Peter J Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 328–339, 1991.

[VRS91] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Zuk01] Ulrich Zukowski. *Flexible Computation of the Well-Founded Semantics of Normal Logic Programs*. PhD thesis, Faculty of Computer Science and Mathematics, University of Passau, 2001.

A Proofs

Proof of Theorem 1. First we show that the founded model is consistent. A given program cannot contain negative facts or negative conclusions, so all negative literals in $\mathit{Founded}(\pi)$ are added by the construction. For a predicate declared uncertain and not complete, no negative literals are added. For a predicate P declared uncertain and complete, consistency follows from the fact that the only rule defining `n.`P in $\mathit{Cmpl}(\pi)$ is the inverse of the only rule defining P in $\mathit{Cmpl}(\pi)$. The body of the former rule is the negation of the body B of the latter rule. Monotonicity of $T_{I_{i-1} \cup \mathit{Proj}(\mathit{NameNeg}(\mathit{Cmpl}(\pi))), S_i)}$ implies that the value of a ground instance of B cannot change from true to false, or *vice versa*, during the fixed point calculation for the SCC S containing P. Using this observation, it is easy to show by induction on the number of iterations of the fixed point calculation for S that an atom for P and its negation cannot both be added to the interpretation. For a certain predicate, consistency follows from the fact that *AddNeg* adds only literals whose complement is not in the interpretation.

Constraint models are consistent by definition. ■

Proof of Theorem 2. First we show that $\mathit{Founded}(\pi)$ is a model of $\mathit{Cmpl}(\pi)$. $\mathit{Founded}(\pi)$ contains all facts in π, because each fact in π is either merged into a combined rule in $\mathit{Cmpl}(\pi)$ or copied unchanged into $\mathit{Cmpl}(\pi)$, and in either case is added to the founded model by the LFP for some SCC. Consider a rule $C \leftarrow B$ in $\mathit{Cmpl}(\pi)$ with predicate Q in the conclusion C. Note that C may be a positive or negative literal. If the body B becomes true before or in the LFP for the SCC S containing Q, then the corresponding disjunct in the combined rule defining Q becomes true before or in that LFP, so the conclusion C is added to the interpretation by that LFP, so the rule is satisfied. It remains to show that B could not become true after that LFP. B cannot become true during processing of a subsequent SCC, because SCCs are processed in dependency order, so subsequent SCCs do not contain predicates in B. We prove by contradiction that B cannot become true in *AddNeg* for S, i.e., we suppose B becomes true in *AddNeg* for S and show a contradiction. *AddNeg* for S adds only negative literals for certain predicates in S, so B must contain such a literal, say $\neg P(\ldots)$. P and Q are in the same SCC S, so P must be defined, directly or indirectly, in terms of Q. Since P is certain and is defined in terms of Q, Q must be certain. Since Q and P are defined in the same SCC S, and Q depends negatively on P, Q has a circular negative dependency, so Q must be uncertain, a contradiction.

Constraint models are 2-valued models of $\mathit{Cmpl}(\pi)$ by definition.

Any model of $\mathit{Cmpl}(\pi)$ is also a model of π, because π is logically equivalent to the subset of $\mathit{Cmpl}(\pi)$ obtained by removing the completion rules added by *AddInv*. ■

Proof of Theorem 3. Since P can be declared certain, predicates in S do not have negative circular dependency, so they can be declared certain. Let π be the program with all predicates in S declared certain. Let $M = \mathit{Founded}(\pi)$. Let π' be the variant of π with all predicates in S declared uncertain and complete. Let $M' = \mathit{Founded}(\pi')$. Let $\prec$ be the restriction to S of the dependency relation between predicates in π. The premise about absence of positive circular dependency implies $(S, \prec)$ is acyclic. We prove by induction on $(S, \prec)$ that, for each predicate Q in S, M and M' contain the same literals for Q.

Recall that the interpretation of Q in M is calculated as follows. In the LFP for the SCC containing Q, atoms for Q are inferred (i.e., added to M) using the facts and rules for Q in π; after that LFP, every remaining atom for Q is set to false by *AddNeg* (i.e., the atom's negation is added to M). Note that M is 2-valued for Q.

Consider the calculation of the interpretation of Q in M'. In the LFP for the SCC containing Q, atoms for Q are inferred using the combined rule for Q, and atoms for `n.`P are inferred using the completion rule for Q. The combined rule for Q is logically equivalent to the facts and rules for Q in π, so M and M' contain the same positive literals for Q. Next, we show that M' (like M) is 2-valued for Q; this and the fact that they contain the same positive literals for Q imply that they also contain the same negative literals for Q, and hence they contain the same literals for Q, as desired. By the induction hypothesis, M' contains the same literals as M for all predicates (if any) on which Q depends, hence M' is 2-valued for those predicates. This implies that, for every ground instance R of the combined rule for Q, either the body of R evaluates to true in M', or the body of the corresponding ground instance R_{comp} of the completion rule for Q (i.e., R_{comp} is the inverse of R) evaluates to true. This implies that M' is 2-valued for Q. ■

Proof of Theorem 4. It suffices to show that, if some predicate in S is uncertain, then all predicates in S are uncertain. Suppose S contains an uncertain predicate P, and let Q be another predicate in S. Q is defined directly or indirectly in terms of predicate P, and P is uncertain, so Q must be uncertain. ■

Proof of Theorem 5. The proof is based on a straightforward correspondence between the constructions of founded semantics of π and $\mathit{Merge}_S(\pi)$.

Note that:

- All predicates in S are certain, or all of them are uncertain, by Theorem 4.
- There is a 1-to-1 correspondence between the set of disjuncts in the bodies of the rules for predicates in S in $\mathit{Cmpl}(\pi)$ and the set of disjuncts in the body of the rule for `holds` in $\mathit{Cmpl}(\mathit{Merge}_S(\pi))$.
- If predicates in S are uncertain and complete, there is a 1-to-1 correspondence between the set of conjuncts in the bodies of the completion rules for predicates in S in $\mathit{Cmpl}(\pi)$ and the set of conjuncts in the body of the completion rule for `holds` in $\mathit{Cmpl}(\mathit{Merge}_S(\pi))$.

Based on these observations, it is straightforward to show that:

- For each predicate P not in S, each atom A for P or `n.`P is derivable in the semantics for π iff A is derivable in the semantics for $\mathit{Merge}_S(\pi)$.

- In the LFP for the SCC containing S, for each predicate P in S, an atom A for P is derivable using a disjunct of the rule for P in $\mathit{Cmpl}(\pi)$ iff $\mathit{MergeAtom}_S(A)$ is derivable using the corresponding disjunct of the rule for `holds` in $\mathit{Cmpl}(\mathit{Merge}_S(\pi))$.
- In the LFP for the SCC containing S, for each uncertain complete predicate P in S, an atom A for `n.`P is derivable using the completion rule for P in π iff $\mathit{MergeAtom}_S(A)$ is derivable using the corresponding conjuncts in the completion rule for `holds` in $\mathit{Merge}_S(\pi)$ (the other conjuncts in the completion rule for `holds` have the form `v` $\neq$ `'`Q`'` $\vee \cdots$ and hence are true when considering derivation of atoms of the form `n.holds('`P`', ...)`).
- In AddNeg for the SCC containing S, for each certain predicate P in S, an atom A for `n.`P is inferred in the semantics for π iff $\mathit{MergeAtom}_S(A)$ is inferred in the semantics for $\mathit{Merge}_S(\pi)$.

■

Proof of Theorem 6. For certain predicates, the program completion *Cmpl* has no effect, and *LFPbySCC* is essentially the same as the definition of stratified semantics, except using SCCs in the dependency graph instead of strata. The SCCs used in founded semantics subdivide the strata used in stratified semantics; intuitively, this is because predicates are put in different SCCs whenever possible, while predicates are put in different strata only when necessary. This subdivision of strata does not affect the result of *LFPbySCC*, so founded semantics is equivalent to the stratified semantics. ■

Proof of Theorem 7. Observe that, for a program π satisfying the hypotheses of the theorem, $\mathit{Cmpl}(\pi)$ is logically equivalent to π. Every constraint model is a 2-valued model of $\mathit{Cmpl}(\pi)$ and hence a 2-valued model of π. Consider a 2-valued model M of π. Since π satisfies the hypotheses of the theorem, $\mathit{Founded}(\pi)$ contains only positive literals, added by the LFPs in *LFPbySCC*. The LFPs add a positive literal to $\mathit{Founded}(\pi)$ only if that literal is implied by the facts and rules in π and therefore holds in all 2-valued models of π. Therefore, $\mathit{Founded}(\pi) \subseteq M$. M satisfies π and hence, by the above observation, also $\mathit{Cmpl}(\pi)$. Thus, M is a constraint model of π. ■

Proof of Theorem 8. Consider an intensional predicate P. It is straightforward to show that the LFP for the SCC containing P using the combined rule for P in $\mathit{Cmpl}(\pi)$, of the form $C \leftarrow B$, and its inverse, of the form $\neg C \leftarrow \neg B$, is equivalent to satisfying the conjunct for P in $\mathit{CCmpl}_D(\pi)$, of the form $C \cong B$. The proof for the forward direction ($\Rightarrow$) of the equivalence is a case analysis on the truth value of the body B in $\mathit{Founded}(\pi)$: (1) if B is true, then the LFP uses the combined rule $C \leftarrow B$ to infer C is true, so $C \cong B$ holds; (2) if B is false, then the LFP uses the inverse rule to infer C is false, so $C \cong B$ holds; (3) if B is undefined, then neither rule applies and C is undefined, so $C \cong B$ holds. Similarly, the proof for the reverse direction ($\Leftarrow$) is a simple case analysis on the truth values of B and C (which are the same, since $C \cong B$ by assumption).

Consider an extensional predicate P. Let S be the set of atoms for P in π. It is easy to show that $\mathit{Founded}(\pi)$ and $\mathit{Fitting}(\pi)$ contain the atoms in S and contain negative literals for P for all other arguments. ■

Proof of Theorem 9. (a) This follows from Theorem 8 and the observation that, if π satisfies the premises of Theorem 8, and π' is obtained from π by changing the declarations of some extensional predicates from certain to uncertain, then $\mathit{Founded}(\pi') \subseteq \mathit{Founded}(\pi)$; intuitively, fewer assumptions are made about uncertain predicates, so $\mathit{Founded}(\pi')$ contains fewer conclusions.

(b) This follows from part (a) and the observation that p is undefined in *Founded*(π), and p is false in *Fitting*(π) (i.e., *Fitting*(π) contains $\neg p$), so the inclusion relation is strict. ■

Proof of Theorem 10. (a) This follows from Theorem 8, the differences between the declarations assumed in Theorem 8 and the default declarations, and the effect of those differences on the founded model. It is easy to show that the default declarations can be obtained from the declarations assumed in Theorem 8 by changing the declarations of some intensional predicates from uncertain and complete to certain. Let P be such a predicate. This change does not affect the set S of positive literals derived for P, because the combined rule for P is equivalent to the original rules and facts for P. This change can only preserve or increase the set of negative literals derived for P, because *AddNeg* derives all negative literals for P that can be derived while preserving consistency of the interpretation (in particular, negative literals for all arguments of P not in S).

(b) This follows from the proof of part (a) and the observation that the additional premise for part (b) implies there is a literal p for P that is undefined in *Fitting*(π) and defined (i.e., true or false) in *Founded*(π) (because *Founded*(π) is 2-valued for P), so the inclusion is strict. ■

Proof of Theorem 11. We prove an invariant that, at each step during the construction of *Founded*(π), the current approximation I to *Founded*(π) satisfies $I \subseteq \mathit{WFS}(\pi)$. First we show, using the induction hypothesis, that literals added to I by the LFPs in *LFPbySCC* are in $\mathit{WFS}(\pi)$. Consider a literal p added by a combined rule $C \leftarrow B$. This implies B is true in I. By the induction hypothesis, $I \subseteq \mathit{WFS}(\pi)$, so B is true in $\mathit{WFS}(\pi)$. Using the rule in π corresponding to a disjunct in B that is true in I, we conclude $p \in T_\pi(\mathit{WFS}(\pi))$. The definition of $\mathit{WFS}(\pi)$ implies $\mathit{WFS}(\pi)$ is closed under T_π, so $p \in \mathit{WFS}(\pi)$. Consider a literal $\neg p$ added by a combined rule $\neg C \leftarrow \neg B$. All of the disjuncts in the negation normal form of B are true in I, so the bodies of all rules in π that derive p are false in I and, by the induction hypothesis, are false in $\mathit{WFS}(\pi)$, so $p \in U_\pi(\mathit{WFS}(\pi))$. The definition of $\mathit{WFS}(\pi)$ implies $\mathit{WFS}(\pi)$ is closed under $\neg \cdot U_\pi$, so $\neg p \in \mathit{WFS}(\pi)$.

It remains to show that negative literals added to I by *AddNeg* are in $\mathit{WFS}(\pi)$. Consider an SCC S in the dependency graph. Let N_S be the set of atoms whose negations are added to I by *AddNeg* for S. Let I_S denote the interpretation produced by the LFP for S. Since U_π is monotone, it suffices to show that N_S is an unfounded set for π with respect to I_S, i.e., for each atom A in N_S, for each ground instance $A \leftarrow B$ of a rule of π with conclusion A, either (1) some hypothesis in B is false in I_S or (2) some positive hypothesis in B is in N_S. We use a case analysis on the truth value of B in I_S. B cannot be true in I_S, because if it were, A would be added to I_S by the LFP and would not be in N_S. If B is false in I_S, then case (1) holds.

Suppose B is undefined in I_S. This implies that at least one hypothesis H in B is undefined in I_S. Let Q be the predicate in A, and let P be the predicate in H. *AddNeg* adds literals only for certain predicates, so Q is certain. Q depends on P, so P must be certain, and P must be in S or a previous SCC. If P were in a previous SCC, then I_S would be 2-valued for P, and H would be T or F in I_S, a contradiction, so P is in S. Since P is in S, and H is undefined in I_S, *AddNeg* adds $\neg H$ to I_S, i.e., H is in N_S. Q is certain, so Q does not have circular negative dependency; therefore, since P and Q are both in S, H must be a positive hypothesis. Thus, case (2) holds. ■

Proof of Theorem 12. Let $M \in$ *Supported*(π). We show $M \in$ *Constraint*(π), i.e., *Founded*$(\pi) \subseteq M$ and M is a 2-valued model of *Cmpl*(π). Theorem 15 of [ABW88] shows that an interpretation I is a supported model of π iff I is a 2-valued model of *CCmpl*(π) . Therefore, M is a model of *CCmpl*(π). Theorem 8 implies that *Founded*(π) is the least model of *CCmpl*(π), so *Founded*$(\pi) \subseteq M$. For each predicate P for which *Cmpl*(π) contains a rule (i.e., each predicate that appears in at least one fact or conclusion in π), the conjunction of the combined rule for P and its inverse in $\mathit{Cmpl}(\pi)$ is logically equivalent for 2-valued models to the equivalence for P in $\mathit{CCmpl}(\pi)$; this is a straightforward tautology in 2-valued logic. Thus, since M is a model of $\mathit{CCmpl}(\pi)$, it is also a model of $\mathit{Cmpl}(\pi)$. Thus, M is a constraint model of π.

Let $M \in \mathit{Constraint}(\pi)$. We show that $M \in \mathit{Supported}(\pi)$. By definition, M is 2-valued, M satisfies $\mathit{Cmpl}(\pi)$, and $\mathit{Founded}(\pi) \subseteq M$. By Apt et al.'s theorem cited above, it suffices to show that M is 2-valued and satisfies $\mathit{CCmpl}(\pi)$. We show that the clause in $\mathit{CCmpl}(\pi)$ for each predicate P is satisfied, by case analysis on P.

Case 1: P does not appear in any fact or conclusion. *AddNeg* makes P false for all arguments in $\mathit{Founded}(\pi)$ and hence in M, so M satisfies the clause for P in $\mathit{CCmpl}_U(\pi)$. Case 2: P appears in some fact or conclusion.

Case 2: P appears in some fact or conclusion. M satisfies $\mathit{Cmpl}(\pi)$, which contains a combined rule of the form $C \leftarrow B$ for P and the inverse rule $\neg C \leftarrow \neg B$. Since M is 2-valued, the conjunction of those rules is equivalent to $C \cong B$, which is the clause for P in $\mathit{CCmpl}_D(\pi)$. ■

Proof of Theorem 13. This theorem follows from Theorem 12, and the observation that, if π satisfies the premises of Theorem 12, and π' is obtained from π by changing the declarations of some extensional predicates from certain to uncertain, then $\mathit{Constraint}(\pi) \subseteq \mathit{Constraint}(\pi')$. To prove this, we analyze how the change in declarations affects $\mathit{Founded}(\pi)$ and $\mathit{Cmpl}(\pi)$, and then show that the changes to $\mathit{Founded}(\pi)$ and $\mathit{Cmpl}(\pi)$ preserve or increase the set of constraint models.

As shown in the proof of Theorem 9, the founded model decreases, i.e., $\mathit{Founded}(\pi') \subseteq \mathit{Founded}(\pi)$. This may allow additional constraint models, because more models satisfy the requirement of being a superset of the founded model. The effect on $\mathit{Cmpl}(\pi)$ is to add completion rules for the predicates whose declaration changed. This does not change the set of constraint models, because all constraint models of π satisfy these completion rules. This conclusion follows from the lemma: For a program π and a certain predicate P in π, every constraint model M of π satisfies the completion rule R for P (even though this rule does not appear in $\mathit{Cmpl}(\pi)$, because P is certain). To see this, first note that P and hence all predicates on which it depends are certain, so all predicates used in R are certain, so $\mathit{Founded}(\pi)$ is 2-valued for those predicates, so $\mathit{Founded}(\pi)$ and M contain the same literals for those predicates, so M satisfies R iff $\mathit{Founded}(\pi)$ satisfies R. To see that $\mathit{Founded}(\pi)$ satisfies R, let $C \leftarrow B$ denote the combined rule for P, so R is $\neg C \leftarrow \neg B$, and note that $\mathit{Founded}(\pi)$ contains a positive literal for P for arguments for which B holds in $\mathit{Founded}(\pi)$ and contains a negative literal for P for all other arguments, and B is false for all of those other arguments, because $\mathit{Founded}(\pi)$ is 2-valued for all predicates used in B and hence for B. ■

Proof of Theorem 14. This theorem follows from Theorem 12, the differences between the declarations assumed in Theorem 12 and the default declarations, and the effect of those differences on the constraint models. We analyze how the differences in declaration affect $\mathit{Founded}(\pi)$ and $\mathit{Cmpl}(\pi)$, and then analyze how the changes to $\mathit{Founded}(\pi)$ and $\mathit{Cmpl}(\pi)$ affect the set of constraint models. Specifically, we show that the set of constraint models is preserved or decreases and hence that $\mathit{Constraint}(\pi) \subseteq \mathit{Supported}(\pi)$.

The declarations assumed in Theorem 12 are the same as in Theorem 8. Recall from the proof of Theorem 10 that the default declarations can be obtained from those declarations by changing the declarations of some intensional predicates from uncertain and complete to certain. Let S be the set of predicates whose declarations change. As discussed in the proof of Theorem 10, the effect of these declaration changes on $\mathit{Founded}(\pi)$ is to preserve or increase the set of negative literals for predicates in S. The effect of these declaration changes on $\mathit{Cmpl}(\pi)$ is to remove completion rules for predicates in S.

Now consider the effects of these changes on the set of constraint models. Adding negative literals to the founded model has the effect of decreasing the set of constraint models of the program, because constraint models not containing those literals are eliminated, since each constraint model must be a superset of the founded model. Removing from $\mathit{Cmpl}(\pi)$ the completion rules for predicates in S does not cause any further changes to the set of constraint models, because the interpretation of predicates in S in the constraint models is now completely determined by the requirement that the constraint models are supersets of the founded model, because the founded model is 2-valued for predicates in S. ■

Proof of Theorem 15. Let $M \in \mathit{SMS}(\pi)$. We need to show $M \in \mathit{Constraint}(\pi)$, i.e., $\mathit{Founded}(\pi) \subseteq M$ and M is a 2-valued model of $\mathit{Cmpl}(\pi)$. By Theorem 11, $\mathit{Founded}(\pi) \subseteq \mathit{WFS}(\pi)$. M is a fixed point of W_π [VRS91, Theorem 5.4], so $\mathit{WFS}(\pi) \subseteq M$. By transitivity, $\mathit{Founded}(\pi) \subseteq M$. It is easy to show that M is a 2-valued model of $\mathit{Cmpl}(\pi)$ iff it is a 2-valued model of π and $\mathit{Cmpl}_N(\pi)$, where $\mathit{Cmpl}_N(\pi)$ denotes the completion rules added by AddInv ("N" reflects the negative conclusions). Gelfond and Lifschitz proved that every stable model of π is a 2-valued model of π [GL88, Theorem 1]. It remains to show that M is a model of $\mathit{Cmpl}_N(\pi)$. Let $\neg P \leftarrow \neg H_1 \wedge \cdots \wedge \neg H_n$ be a rule in $\mathit{Cmpl}_N(\pi)$. We need to show that, if M satisfies $\neg H_1 \wedge \cdots \wedge \neg H_n$, then M satisfies $\neg P$. It suffices to show $P \in U_\pi(M)$, because this implies $\neg P \in M$. The rules defining P in π have the form $P \leftarrow H_i$, for $i \in [1..n]$, and each H_i is false in M by assumption, so some conjunct in each H_i is false in M, so by definition of unfounded set, $P \in U_\pi(M)$. ■

Proof of Theorem 17. First, we show that $\mathit{FoundedClosed}(\pi)$ is 2-valued for predicates in S. Let RS be the set of all instances of combined rules and completion rules in $\mathit{Cmpl}(\pi)$ for predicates in S. Note that every positive literal and negative literal for every predicate in S appears as the conclusion of at least one rule in G (this holds even if rules in π contain conclusions of the form `p(x,x)` or `p(x,0)`, due to the fresh variables and existential quantifiers introduced by $\mathit{Combine}$). Let UA be the set of atoms for predicates in S that are undefined in $\mathit{Founded}(\pi)$. For each atom A in UA, since the predicate in A is complete and A is not T or F in $\mathit{Founded}(\pi)$, (1) for every rule R in RS with conclusion A, some hypothesis of R is F or U in $\mathit{Founded}(\pi)$, and (2) for some rule R in RS with conclusion A, some hypothesis of R is U in $\mathit{Founded}(\pi)$. Since all predicates in SCCs that precede S are certain, these undefined hypotheses must be atoms in UA or their negations. Define a dependence relation $\rightarrow$ on UA by: $B \rightarrow A$ if some rule R in RS with conclusion A has an undefined hypothesis that is B or $\neg B$. The previous observation implies that, for every A in UA, there exists B in UA such that $B \rightarrow A$. Since UA is finite, this implies that every atom in UA is in a $\rightarrow$-cycle. Since predicates in S do not have circular negative dependency, this implies that all hypotheses involved in the cycle are positive. These observations, together with all predicates in S being closed, imply that the literals in every cycle, and hence all atoms in UA, are in $\mathit{SelfFalse}_\pi(\mathit{Founded}(\pi))$. This implies that $\mathit{FoundedClosed}(\pi)$ contains the negations of all literals in UA. Therefore, $\mathit{FoundedClosed}(\pi)$ is 2-valued.

Next, we show that $\mathit{FoundedClosed}(\pi) = \mathit{FoundedClosed}(\pi')$. For each predicate P in S, the two programs contain equivalent rules for adding positive literals for P to the founded model, because the combined rule for P in π is logically equivalent to the original rules for P in π, so $\mathit{FoundedClosed}(\pi)$ and $\mathit{FoundedClosed}(\pi')$ contain the same positive literals for P. Since both models are 2-valued for P, they also contain the same negative literals for P.

Finally, we show that $\mathit{ConstraintClosed}(\pi) = \mathit{ConstraintClosed}(\pi')$. We consider the three conditions in the definition of $\mathit{ConstraintClosed}$, in turn.

Consider the first condition, namely, $\mathit{FoundedClosed}(\pi) \subseteq M$. It is equivalent for the two programs, because they have the same founded model.

Consider the second condition, namely, M satisfies $\mathit{Cmpl}(\pi)$. It differs for the two programs in that $\mathit{Cmpl}(\pi)$ contains combined rules and completion rules for predicates in S, while $\mathit{Cmpl}(\pi')$ contains the original rules in π for predicates in S. Since $\mathit{FoundedClosed}(\pi) = \mathit{FoundedClosed}(\pi')$, Theorem 1 implies $\mathit{FoundedClosed}(\pi)$ is a model of $\mathit{Cmpl}(\pi)$ and $\mathit{Cmpl}(\pi')$. Since $\mathit{FoundedClosed}(\pi)$ is 2-valued for predicates in S and all predicates on which they depend, it is 2-valued for all predicates used in those rules. Therefore, every M satisfying $\mathit{FoundedClosed}(\pi) \subseteq M$ contains the same literals as $\mathit{FoundedClosed}(\pi)$ for all predicates used in those rules, so M satisfies $\mathit{Cmpl}(\pi)$ and $\mathit{Cmpl}(\pi')$. Thus, the second condition is equivalent for the two programs.

Consider the third condition, namely, $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq M$. $\mathit{FoundedClosed}(\pi)$ is 2-valued for predicates in S and all predicates on which they depend, so for every M satisfying $\mathit{FoundedClosed}(\pi) \subseteq M$, $\mathit{FoundedClosed}(\pi)$ and M contain the same literals for those predicates, so $\mathit{SelfFalse}_\pi(M) = \mathit{SelfFalse}_\pi(\mathit{FoundedClosed}(\pi))$ and $\mathit{SelfFalse}_{\pi'}(M) = \mathit{SelfFalse}_{\pi'}(\mathit{FoundedClosed}(\pi))$. For every instance R of a rule whose conclusion is for a predicate in S, every hypothesis of R is T or F (not undefined) in $\mathit{FoundedClosed}(M)$, so disjunct (2) in the definition of self-false set cannot be used to treat any additional hypothesis of R as F, regardless of which predicates are closed, so $\mathit{SelfFalse}_\pi(\mathit{FoundedClosed}(\pi)) = \mathit{SelfFalse}_{\pi'}(\mathit{FoundedClosed}(\pi))$. Using these equalities and transitivity, $\mathit{SelfFalse}_\pi(M) = \mathit{SelfFalse}_{\pi'}(M)$. Thus, the third condition is equivalent for the two programs. ■

Proof of Theorem 18. First, we show $\mathit{FoundedClosed}(\pi) \subseteq \mathit{WFS}(\pi)$, by proving by induction on the computation of the least fixed point that, in each step, $F_\pi(I) \subseteq \mathit{WFS}(\pi)$. The induction hypothesis is $I \subseteq \mathit{WFS}(\pi)$, and we need to show $F_\pi(I) \subseteq \mathit{WFS}(\pi)$. It suffices to show (1) $\mathit{Founded}(\pi \cup I) \subseteq \mathit{WFS}(\pi)$ and (2) $\neg \cdot \mathit{SelfFalse}_\pi(\mathit{Founded}(\pi \cup I)) \subseteq \mathit{WFS}(\pi)$, since $F_\pi(I)$ is the union of these two sets.

Proof of (1): By Theorem 11, $\mathit{Founded}(\pi \cup I) \subseteq \mathit{WFS}(\pi \cup I)$. It is easy to show that, for any subset I of $\mathit{WFS}(\pi)$, $\mathit{WFS}(\pi \cup I) = \mathit{WFS}(\pi)$. Thus, $\mathit{Founded}(\pi \cup I) \subseteq \mathit{WFS}(\pi)$.

Proof of (2): $\mathit{SelfFalse}_\pi$ is monotone, and $\mathit{Founded}(\pi \cup I) \subseteq \mathit{WFS}(\pi)$ from (1), so $\mathit{SelfFalse}_\pi(\mathit{Founded}(\pi \cup I)) \subseteq \mathit{SelfFalse}_\pi(\mathit{WFS}(\pi))$. $\mathit{SelfFalse}_\pi$ is U_π restricted to specified predicates, so $\mathit{SelfFalse}_\pi(I) \subseteq U_\pi(I)$ for any interpretation I. Thus, $\mathit{SelfFalse}(\mathit{Founded}(\pi \cup I)) \subseteq U_\pi(\mathit{WFS}(\pi))$. By definition of WFS_π, $\neg \cdot U_\pi(\mathit{WFS}(\pi)) \subseteq \mathit{WFS}(\pi)$. Thus, $\neg \cdot \mathit{SelfFalse}(\mathit{Founded}(\pi \cup I)) \subseteq \mathit{WFS}(\pi)$.

Second, we show $\mathit{WFS}(\pi) \subseteq \mathit{FoundedClosed}(\pi)$, by proving by induction on the computation of the least fixed point that, in each step, $W_\pi(I) \subseteq \mathit{FoundedClosed}(\pi)$. The induction hypothesis is $I \subseteq \mathit{FoundedClosed}(\pi)$, and we need to show $W_\pi(I) \subseteq \mathit{FoundedClosed}(\pi)$. It suffices to show (a) $T_\pi(I) \subseteq \mathit{FoundedClosed}(\pi)$ and (b) $\neg \cdot U_\pi(I) \subseteq \mathit{FoundedClosed}(\pi)$, since $W_\pi(I)$ is the union of these two sets.

The proof of (a) is straightforward using the induction hypothesis and the definitions of T_π and *Founded*. The proof of (b) uses the following lemma relating unfounded sets and self-false sets, which is easily proved from the definitions: $U_\pi(I) \subseteq \mathit{SelfFalse}_\pi(I)$ when all predicates are uncertain, complete, and closed. The induction hypothesis for (b) is $I \subseteq \mathit{FoundedClosed}(\pi)$. U_π is monotone, so $U_\pi(I) \subseteq U_\pi(\mathit{FoundedClosed}(\pi))$. The above lemma implies $U_\pi(\mathit{FoundedClosed}(\pi)) \subseteq \mathit{SelfFalse}_\pi(\mathit{FoundedClosed}(\pi))$. By transitivity, $U_\pi(I) \subseteq \mathit{SelfFalse}_\pi(\mathit{FoundedClosed}(\pi))$. By definition of *FoundedClosed*, $\neg \cdot \mathit{SelfFalse}_\pi(\mathit{FoundedClosed}(\pi)) \subseteq \mathit{FoundedClosed}(\pi)$. Using this inequality, the preceding inequality, and transitivity, we conclude $\neg \cdot U_\pi(I) \subseteq \mathit{FoundedClosed}(\pi)$. ■

Proof of Theorem 19. Proof that $\mathit{SMS}(\pi) \subseteq \mathit{ConstraintClosed}(\pi)$. The proof uses Theorem 15, which has the additional hypothesis that all predicates have default declarations as certain or uncertain. Theorem 15 is applicable here nevertheless, because that additional hypothesis is unnecessary in the context of the other hypotheses of this theorem. To see this, note that non-default declarations of predicates as certain or uncertain can differ from the default declarations only by unnecessarily declaring some predicates uncertain. By hypothesis, those predicates are also declared complete and closed. By applying Theorem 17 to each SCC containing those predicates, we conclude that these non-default declarations do not change the founded and constraint semantics. Therefore, we can assume in the remainder of the proof that predicates have their default declarations as certain or uncertain.

Let $M \in \mathit{SMS}(\pi)$. We need to show $M \in \mathit{ConstraintClosed}(\pi)$, i.e., (1) $\mathit{FoundedClosed}(\pi) \subseteq M$, (2) M is a model of $\mathit{Cmpl}(\pi)$, and (3) $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq M$.

Proof of (1): As shown in the proof of Theorem 15, $\mathit{WFS}(\pi) \subseteq M$. By Theorem 18, $\mathit{FoundedClosed}(\pi) = \mathit{WFS}(\pi)$, so $\mathit{FoundedClosed}(\pi) \subseteq M$. Proof of (2): Same as in the proof of Theorem 15. Proof of (3): Every uncertain predicate is closed, so $\mathit{SelfFalse}_\pi(M) \subseteq U_\pi(M)$, so $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq W_\pi(M)$. M is a fixed point of W_π [VRS91, Theorem 5.4], so $W_\pi(M) = M$. Using this to simplify the right side of the previous inequality, we conclude $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq M$.

Proof that $\mathit{ConstraintClosed}(\pi) \subseteq \mathit{SMS}(\pi)$. Let $M \in \mathit{ConstraintClosed}(\pi)$. We need to show $M \in \mathit{SMS}(\pi)$; this is equivalent to showing M is a fixed point of W_π [VRS91, Theorem 5.4]. We prove $W_\pi(M) \subseteq M$ and $M \subseteq W_\pi(M)$.

Proof that $W_\pi(M) \subseteq M$: We need to show $T_\pi(M) \subseteq M$ and $\neg \cdot U_\pi(M) \subseteq M$. The former follows from the fact that M is a model of $\mathit{Cmpl}(\pi)$.

The latter follows from $\neg \cdot \mathit{SelfFalse}_\pi(M) \subseteq M$ and $\mathit{SelfFalse}_\pi(M) = U_\pi(M)$, as shown next. Since the definition of $\mathit{SelfFalse}_\pi(M)$ is obtained from the definition of $U_\pi(M)$ by limiting in the recursive disjunct to closed predicates, $\mathit{SelfFalse}_\pi(M) \subseteq U_\pi(M)$ always holds. Since all uncertain predicates are also closed, to show $\mathit{SelfFalse}_\pi(M) = U_\pi(M)$, it suffices to show that, for each atom A in $U_\pi(M)$ for a certain predicate P, $A \in \mathit{SelfFalse}_\pi(M)$. To see this, note that $\neg A \in \mathit{FoundedClosed}_\pi(M)$, because $\mathit{FoundedClosed}(M)$ includes all negative literals for certain predicates that can be in any consistent model of π (recall that certain predicates cannot depend on uncertain predicates, so this holds regardless of undefined values in $\mathit{FoundedClosed}(M)$). Since $\neg A \in \mathit{FoundedClosed}_\pi(M)$ and $\mathit{FoundedClosed}_\pi(M) \subseteq M$, we have $\neg A \in M$ and hence $A \in \mathit{SelfFalse}_\pi(M)$.

Proof that $M \subseteq W_\pi(M)$: Consider any literal in M. We need to show that the literal is in $W_\pi(M)$.

Case 1: Consider a positive literal A in M. We show $A \in T_\pi(M)$ hence $A \in W_\pi(M)$.

Case 1.1: A is for a certain predicate. $\mathit{FoundedClosed}(M)$ and M contain the same literals for such predicates, so $A \in \mathit{FoundedClosed}(M)$, so $A \in T_\pi(I)$, where I is the intermediate approximation to $\mathit{FoundedClosed}(M)$ at the step when A is added. T_π is monotonic, and $I \subseteq \mathit{FoundedClosed}(M) \subseteq M$, so $A \in T_\pi(M)$.

Case 1.2: A is for a uncertain predicate P. M satisfies $\mathit{Cmpl}(\pi)$. $\mathit{Cmpl}(M)$ contains the combined rule $C \leftarrow B$ for P and its inverse $\neg C \leftarrow \neg B$. M is 2-valued, and in 2-valued models, the conjunction of these two rules is equivalent to $C \Leftrightarrow B$. Therefore, A is derivable in M using an instance of the combined rule for P, which is logically equivalent to the original rules for P in π, so

$A \in T_\pi(M)$.

Case 2: consider a negative literal $\neg A$ in M. We show $A \in U_\pi(M)$ hence $\neg A \in W_\pi(M)$.

Case 2.1: $\neg A$ is for a certain predicate. $\mathit{FoundedClosed}(M)$ and M contain the same literals for such predicates, so $\neg A \in \mathit{FoundedClosed}(\pi)$. By Theorem 18, $\mathit{FoundedClosed}(\pi) = \mathit{WFS}(\pi)$, so $\neg A \in \mathit{WFS}(\pi)$, so $A \in U_\pi(\mathit{FoundedClosed}(\pi))$, so by monotonicity, $A \in U_\pi(M)$.

Case 2.2: $\neg A$ is for a uncertain predicate P. By reasoning similar to case 1.2, $\neg A$ is derivable in M using an instance of the inverse of the combined rule for P. By definition of the combined rule, this implies that, for every instance with conclusion A of a rule for P in π, the body of the rule evaluates to false in M. This implies $A \in U_\pi(M)$. ■