Recursive Rules with Aggregation: A Simple Unified Semantics*

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

Abstract

Complex reasoning problems are most clearly and easily specified using logical rules, but require recursive rules with aggregation such as count and sum for practical applications. Unfortunately, the meaning of such rules has been a significant challenge, leading to many disagreeing semantics.

This paper describes a unified semantics for recursive rules with aggregation, extending the unified founded semantics and constraint semantics for recursive rules with negation. The key idea is to support simple expression of the different assumptions underlying different semantics, and orthogonally interpret aggregation operations using their simple usual meaning. We present a formal definition of the semantics, prove important properties of the semantics, and compare with prior semantics. In particular, we present an efficient inference over aggregation that gives precise answers to all examples we have studied from the literature. We also apply our semantics to a wide range of challenging examples, and show that our semantics is simple and matches the desired results in all cases. Finally, we describe experiments on the most challenging examples, exhibiting unexpectedly superior performance over well-known systems when they can compute correct answers.

1 Introduction

Many computation problems, including complex reasoning problems in particular, such as program analysis, networking, and decision support, are most clearly and easily specified using logical rules [Liu18]. However, such reasoning problems in practical applications, especially for large applications and when faced with uncertain situations, require the use of recursive rules with aggregation such as count and sum. Unfortunately, the meaning of such rules has been challenging and remains a subject with significant complication and disagreement.

As a simple example, consider a single rule for Tom to attend the logic seminar: "Tom will attend the logic seminar if the number of people who will attend it is at least 20." What does the rule mean? If 20 or more other people will attend, then surely Tom will attend. If only 10 others will attend, then Tom will not attend. What if only 19 other people will attend? Will Tom attend, or not? Although simple, this example already shows that, when aggregation is used in recursive rules—here count is used in a rule that defines "will attend" using "will attend"—the semantics can be tricky.

Some might say that this statement about Tom is ambiguous or ill-specified. However, it is a statement allowed by logic rule languages with predicates, sets, and counts. For example, let predicate `will_attend(p)` denote that `p` will attend the logic seminar; then the statement can be written

*This work was supported in part by NSF under grants CCF-1954837, CCF-1414078, and IIS-1447549 and by ONR under grants N00014-20-1-2751 and N00014-21-1-2719.

as `will_attend(Tom) if count {p: will_attend(p)}` $\geq$ `20`. So the statement must be given a meaning. Indeed, "ambiguous" is a possible meaning, indicating there are two or more answers, and "ill-specified" is another possible meaning, indicating there is no answer. So which one should it be? Are there other possible meanings?

In deductive databases, to avoid challenging cases of aggregation as well as negation, processing of recursive rules with aggregation is largely limited to monotonic programs, i.e., adding a new fact used in a hypothesis cannot make a conclusion change from true to false. However, note that the rule about Tom attending the logic seminar is actually monotonic: adding `attend(p)` for a new `p` can not make the conclusion change from true to false. So, even restricted deductive databases must give a meaning to this rule. What should it be?

In fact, the many different semantics of recursive rules with aggregation are more complex and trickier than even semantics of recursive rules with negation. The latter was already challenging for over 120 years, going back at least to Russell's paradox, for which self-reference with negation is believed to form vicious circles [ID20]. Many different semantics, which disagree with each other, have been studied for recursive rules with negation, as summarized in Section 9. Two of them, well-founded semantics (WFS) [VRS91, VG93] and stable model semantics (SMS) [GL88], became dominant since about 30 years ago.

Semantics of recursive rules with aggregation has been studied continuously since about 30 years ago, and more intensively in more recent years, as discussed in Section 9, especially as they are needed in graph analysis and machine learning applications. However, the many different semantics proposed, e.g., [VG92, GZ19], are even more intricate than WFS and SMS for recursive rules with negation, and include even different extensions for WFS, e.g., [KS91, VG92, SSRB93], and for SMS, e.g., [KS91, MR04, GZ19]. Some authors also changed their own minds about the desired semantics, e.g., [Gel02, GZ19]. Such intricate and disagreeing semantics would be too challenging to use correctly.

This paper describes a simple unified semantics for recursive rules with aggregation as well as negation and quantification. The semantics is built on and extends the founded semantics and constraint semantics of logical rules with negation and quantification developed recently by Liu and Stoller [LS18, LS20a]. The key idea is to capture, and to express in a simple way, the different assumptions underlying different semantics, and orthogonally interpret aggregation operations using their simple usual meaning. We present formal definition of the semantics and prove important properties of the semantics.

In particular, we present an efficient derivability relation for comparisons containing aggregations; it can be computed in linear time and gives precise answers on all examples we have studied from the literature. We also compare with main prior semantics for rules with aggregation, and show how our semantics is direct and follows precisely from usual meanings of aggregations.

We further apply our semantics to a wide range of challenging examples, and show that our semantics is simple and matches the desired semantics in all cases. In particular, our examples include the longest and most sophisticated ones from dozens of previously studied examples [GZ19], where answer sets were obtained by all possible guesses followed by computing sophisticated reduct for each; we show that all the answer sets can be obtained by a simple default assumption and simple least fixed-point computations, as is usual for inductive definitions and commonsense reasoning.

Finally, we describe experiments on three most challenging examples. For one of them, our systematically incrementalized implementation in Python exhibits unexpectedly superior performance over four other most well-known systems. For the other two, we discovered many incorrect results computed by two systems, while the other two systems cannot compute the desired semantics at all. The former two systems have since found and fixed bugs that caused the incorrect results in one of the examples, but developers of these systems also found that fundamentally different inference would be needed to

compute correctly at all for the other example.

The rest of the paper is organized as follows. Section 2 presents the problems and an overview of the solutions. Sections 3 describes the language of recursive rules with unrestricted negation, quantification, and aggregation. Sections 4 and 5 present formal definition of the semantics and prove properties of the semantics. Section 6 discusses and compares with prior semantics. Section 7 illustrates our semantics on a wide range of examples. Section 8 describes experimental results. Section 9 discusses related work and concludes. Appendix A contains additional proofs.

This paper is an extended and revised version of Liu and Stoller [LS22]. The main changes are (1) new Section 5 on properties of the semantics, extending a summary paragraph to include all theorems with proofs, including additional proofs in Appendix A; (2) new Section 6 that discusses and compares with prior semantics in more detail; (3) extended Section 7 on examples, with additional examples 1-4 and 7; and (4) new Section 8 on experiments, extending a summary description about two examples to full descriptions about three examples.

2 Problem and solution overview

The semantics of recursion with negation and aggregation is challenging for several reasons. First, recursion involves self-referencing and cyclic reasoning, for which it is already non-trivial to properly start and finish. Then, negation in recursion incurs self-denying and conflict in cyclic reasoning, which can lead to contradiction, as in Russell's paradox. Finally, aggregation generalizes negation to give rise to even greater challenges in recursion, because a negation essentially corresponds to only the simple case of a count being zero.

The first reason alone already called for a least fixed point semantics, which is beyond first-order logic. The second reason led to major different semantics that are sophisticated and disagreeing when trying to solve conflicts differently. The third reason exacerbated the sophistication and variety to tackle the even greater challenges.

A smallest example. Consider the following recursive rule with aggregation. It says that `p` is true for value `a` if the number of `x`'s for which `p` is true equals `1`:

```
p(a) ← count {x: p(x)} = 1
```

This rule is recursive because inferring a conclusion about `p` requires using `p` in a hypothesis. It uses an aggregation of `count` over a set. While each of recursion and aggregation by itself has a simple meaning, allowing recursion with aggregation is tricky, because recursion is used to define a predicate, which is equivalent to a set, but aggregation using a set requires the set to be already defined.

We use this example in addition to our Tom example in Section 1, for two reasons. First, this and similar small examples are used for comparisons in previous papers, e.g., [FPL11, GZ14, GZ18]. Second, this example differs from the Tom example in that the comparison with count in this example is non-monotonic, i.e., adding more `x`'s for which `p(x)` is true can change the value of the comparison, and thus the conclusion, from true to false; using only one example is insufficient to show the main different cases.

- **Two models: Kemp-Stuckey 1991, Gelfond 2002.** According to Kemp and Stuckey [KS91] and Gelfond [Gel02], the above rule has two models: one empty model, i.e., a model in which nothing is true and thus `p(a)` is false, and one containing only `p(a)`, i.e., `p(a)` is true and everything else is false.

- **One model: Faber et al. 2011, Gelfond-Zhang 2014-2019.** According to Faber, Pfeifer, and Leone [FPL11] and Gelfond and Zhang [GZ14, Examples 2 and 7], [GZ18, Examples 4 and 6], and [GZ19, Example 9], the rule above has only one model: the empty model.

As one of the several main efforts investigating aggregation, Gelfond and Zhang [Gel02, GZ14, ZR16, GZ17, GZ18, GZ19] have studied the challenges and solutions extensively, presenting dozens of definitions and propositions and discussing dozens of examples [GZ19]. Their examples where count is used in inequalities, greater than, etc., with additional variables, with more hypotheses in a rule, or with more rules and facts, are even more complicated. We discuss their most extensive examples in Section 7.

Extending founded semantics and constraint semantics for aggregation. Aggregation, such as count, is a simple concept that even kids understand. So it is stunning to see so many sophisticated treatments for figuring out its meaning when it is used in rules, and to see the many disagreeing semantics resulting from those.

We develop a simple and unified semantics for rules with aggregation as well as negation and quantification by building on founded semantics and constraint semantics [LS18, LS20a] for rules with negation and quantification. The key insight is that disagreeing complex semantics for rules with aggregation are because of different underlying assumptions, and these assumptions can be captured using the same simple binary declarations about predicates as in founded semantics and constraint semantics but generalized to include the meaning of aggregation.

Certain. First, if there is no potential non-monotonicity, including no aggregation in recursion, then the predicate in the conclusion can be declared "certain".

Being certain means that assertions of the predicate are given true or inferred true by simply following rules whose hypotheses are given or inferred true, and the remaining assertions of the predicate are false. This is both the founded semantics and constraint semantics.

For the Tom example, there is no potential non-monotonicity; with this declaration, when given that only 19 others will attend, the hypothesis of the rule is not true, so the conclusion cannot be inferred. Thus Tom will not attend.

Uncertain. Regardless of monotonicity, a predicate can be declared "uncertain". It means that assertions of the predicate can be given or inferred true or false using what is given, and any remaining assertions of the predicate are undefined. This is the founded semantics.

If there are undefined assertions from founded semantics, all combinations of true and false values are checked against the rules and declarations as constraints, yielding a set of possible satisfying combinations. This is the constraint semantics.

Complete, or not complete. An uncertain predicate can be further declared "complete" or not.

Being complete means that all rules that can conclude assertions of the predicate are given. Thus a new rule, called completion rule, can be created to infer negative assertions of the predicate when none of the given rules apply.

Being not complete means that negative assertions cannot be inferred using completion rules, and thus all assertions of the predicate that were not inferred to be true are undefined.

For the Tom example, the completion rule implies: Tom will not attend the logic seminar if the number of people who will attend it is less than 20.

When given that only 19 others will attend, due to the uncertainty of whether Tom will attend,

neither the given rule nor the completion rule will fire. So whether one uses the declaration of complete or not, there is no way to infer that Tom will attend, or Tom will not attend. So, founded semantics says it is undefined.

Then constraint semantics tries both for it to be true, and for it to be false; both satisfy the rule, so there are two models: one where Tom will attend, and one where Tom will not attend.

Closed, or not closed. Finally, an uncertain complete predicate can be further declared "closed" or not.

Being closed means that an assertion of the predicate is made false if inferring it to be true requires itself to be true.

Being not closed means that such assertions are left undefined.

For the Tom example, with this declaration, if there are only 19 others attending, then Tom will not attend in both founded semantics and constraint semantics. This is because inferring that Tom will attend requires Tom himself to attend to make the count to be 20, so it should be made false, meaning that Tom will not attend.

Note that this is the same result as using "certain". Because the rule for deciding whether Tom will attend has no potential non-monotonicity, using "certain" is much simpler and has the same meaning as using "closed", as shown in general in Section 5.

For the smallest example about `p` near the beginning of this section, the equality comparison is not monotonic. Thus `p` must be declared uncertain. This example also shows different semantics when using the declarations of not complete and complete, unlike the Tom example.

- **Not complete.** Suppose `p` is declared not complete. Founded semantics does not infer `p(a)` to be true using the given rule because `count {x: p(x)} = 1` cannot be determined to be true, and nothing infers `p(a)` to be false. Thus `p(a)` is undefined. So is `p(b)` for any constant `b` other than `a` because nothing infers `p(b)` to be true or false. Constraint semantics gives a set of models, each for a different combination of true and false values of `p(c)` for different constants `c` such that the combination satisfies the given rule. This corresponds to what is often called open-world assumption and used in commonsense reasoning.
- **Complete.** Suppose `p` is declared complete but not closed. A completion rule is first added. The precise completion rule is:

  $\neg$ `p(x)` $\leftarrow$ `x` $\neq$ `a` $\vee$ `count {x: p(x)}` $\neq$ `1`

  Founded semantics does not infer `p(a)` to be true or false using the given rule or completion rule, because `count {x: p(x)}` $\neq$ `1` also cannot be determined to be true. Thus `p(a)` is undefined. Founded semantics infers `p(b)` for any constant `b` other than `a` to be false using the completion rule. Constraint semantics gives two models: one with `p(a)` being true, and `p(b)` being false for any constant `b` other than `a`; and one with `p(c)` being false for every constant `c`. This is the same as the two-model semantics per Kemp-Stuckey 1991 and Gelfond 2002.
- **Closed.** Supposed `p` is declared complete and closed. Both founded semantics and constraint semantics give only the second model above, i.e., `p(c)` is false for every constant `c`. They have `p(a)` being false because inferring `p(a)` to be true requires `p(a)` itself to be true. This is the same as the one-model semantics per Faber et al. 2011 and Gelfond-Zhang 2014-2019.

We see that simple binary declarations of the underlying assumptions, with simple inference following

rules and taking rules as constraints, give the different desired semantics.

Relationship with prior semantics. Table 1 summarizes relationships between our unifying semantics and major prior semantics. With different predicate declarations capturing different underlying assumptions, founded semantics and constraint semantics for rules with aggregation extend different prior semantics for rules with negation uniformly, as shown in Table 1 left and middle columns. These extend the matching relationships proved for rules with negation in [LS18, LS20a]. All these relationships are when all predicates in a program have the same declarations, but our founded semantics and constraint semantics also allow different predicates to have different declarations.

Among many different prior semantics for rules with aggregations, there are even different extensions for the same prior semantics for rules with negation, as shown in the right column in Table 1. Unfortunately, most of them are defined for limited cases, or add some case-specific definitions. In particular, simple formal explanations for the disagreements, including among all different extensions for each of WFS and SMS, are completely missing. We are only aware of comparisons by examples or very restricted cases, even for disagreeing semantics by the same authors. However, for all such examples and cases we examined, we found that the desired results for them correspond to our semantics under some appropriate declarations for some predicates. These results are described in Section 6.

Declarations	Semantics	Extending	Reference	Prior Extensions
certain	Founded, Constraint	Stratified (Perfect)	Van Gelder 1986 [VG86]	e.g., [MPR90, RS92]
uncertain, not complete	Founded	(none found)		(none found)
	Constraint	First-Order Logic		e.g., [HLNW01]
uncertain, complete, not closed	Founded	Fitting (Kripke-Kleene)	Fitting 1985 [Fit85]	Pelov et al. 2007 [PDB07]
	Constraint	Supported	Apt et al. 1988 [ABW88]	Pelov et al. 2007 [PDB07]
uncertain, complete, closed	Founded	WFS	Van Gelder et al. 1988 [VGRS88, VRS91]	Kemp-Stuckey 1991 [KS91]
				Van Gelder 1992 [VG92]
				Pelov et al. 2007 [PDB07]
	Constraint	SMS	Gelfond-Lifschitz 1988 [GL88]	Kemp-Stuckey 1991 [KS91]
				Pelov et al. 2007 [PDB07]
				Faber et al. 2011 [FPL11]
				Gelfond-Zhang 2014-2019 [GZ19]

Table 1: Founded semantics and constraint semantics for rules with aggregation with different declarations (for all predicates in a program), extending prior semantics for rules with negation, and prior extensions.

3 Language

We consider Datalog rules extended with unrestricted negation, disjunction, quantification, aggregation, and comparison containing aggregation.

Domain. The *domain* of a program is the set of values that variables can be instantiated with. These values are called *constants*. The domain includes the values that appear in the program and a set *Num* of numbers. *Num* is a bounded range of numbers determined by a *numeric representation bound NRB* and

a *numeric representation precision NRP*, i.e., *Num* contains all numbers in the range $[-NRB, NRB]$ with at most *NRP* decimal places. Numbers with more than *NRP* decimal places that appear in the program or arise during evaluation can be rounded to *NRP* decimal places (such rounding can be reported), or a higher-precision representation can be used.

This rounding or increasing precision is not shown explicitly in the semantics, because the rule language in this paper (which includes all major aggregation operations: count, max, min, and sum) does not include numeric operations that increase the number of decimal places. We use an *NRP* that is at least the maximum number of decimal places in numbers that appear in the program, so all numeric computations are exact, and rounding or increasing precision is not needed.

Datalog rules with unrestricted negation. We first present a simple core form of rules and then describe additional constructs that can appear in rules. The *core form* of a rule is the following, where any P_i may be preceded with $\neg$:

$$Q(X_1, ..., X_a) \leftarrow P_1(X_{11}, ..., X_{1a_1}) \wedge ... \wedge P_h(X_{h1}, ..., X_{ha_h})$$

Symbols $\leftarrow$, $\wedge$, and $\neg$ indicate backward implication, conjunction, and negation, respectively. Q and P_i's are predicates, and each argument X_k and X_{ij} is a constant or a variable. In examples, we use numbers and quoted strings for constants and letters for variables.

If $h = 0$, then there are no P_i's or X_{ij}'s, and $Q(X_1, ..., X_a)$ is called a *fact*. Facts are first transformed so that each X_k that is a variable is instantiated with all constants in the domain of the program. So facts have only constants as arguments for the rest of the paper. Also, for the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case the left side of the backward implication is called the *conclusion*, the right side is called the *body*, and each conjunct in the body is called a *hypothesis*. We do not require variables in the conclusion to be in the hypotheses; it is not needed because rules are used with variables replaced by constants, and the domain of variables is finite.

Disjunction. In a rule body, hypotheses may be combined using disjunction ($\vee$) as well as conjunction. Conjunction and disjunction may be nested arbitrarily.

mixed with conjunction and disjunction.

Quantification. A hypothesis in a rule body can be an existential or universal quantification of the form

$$\begin{array}{ll} \exists\ X_1, ..., X_a \mid B & \text{existential quantification} \\ \forall\ X_1, ..., X_a \mid B & \text{universal quantification} \end{array}$$

where each X_i is a variable in B, and B has the same form as a rule body. Note that this recursive definition allows nested quantifications. Each quantified variable X_i ranges over the domain of the program. The quantifications return true iff for some or all, respectively, combinations of values of $X_1, ..., X_a$, the body B is true.

To restrict a quantified variable X_i to a particular set of values, a user can introduce a predicate D with a fact $D(x)$ for each element x of that set, and write an existential quantification by adding a conjunct $D(X_i)$ to the body and write a universal quantification by adding a disjunct $\neg D(X_i)$ to the body.

Aggregation and comparison. A *set expression* has the form $\{X_1, ..., X_a : B\}$, where each X_i is a variable in B, and the body B has the same form as a rule body. The *arity* of this set expression is a. The body of each set expression is first rewritten to have the same form as the body of a core-form rule, by introducing auxiliary predicates, e.g., `{Y: ∃ X | p(X,Y)}` is rewritten to `{Y: q(Y)}` together with

a new rule `q(Y)` $\leftarrow \exists$ `X | p(X,Y)`, where `q` is a fresh predicate. Each auxiliary predicate is declared complete (which is the default, defined below), except that it is declared closed if some predicate in the body of the rule defining the auxiliary predicate is declared closed. This exception preserves any cycle of dependencies among closed predicates.

An *aggregation* has the form $agg\, S$, where agg is an aggregation operator (`count`, `max`, `min`, or `sum`), and S is a set expression. The aggregation returns the result of applying the respective agg operation (cardinality, maximum, minimum, or sum) to the set value of S. `max` and `min` use the order on numbers, extended lexicographically to an order on tuples. `sum` is on numbers, and on tuples whose first components are numbers; in the latter case, the first components are summed. Note that `count` and `sum` applied to the empty set equal 0, while `max` and `min` applied to the empty set give an error.

A hypothesis of a rule may be a *comparison* of the form

$$agg\, S \odot k \qquad \text{or} \qquad agg\, S \odot agg'\, S'$$

where $agg\, S$ and $agg'\, S'$ are aggregations, the comparison operator $\odot$ is an equality ($=$) or inequality ($\neq$, $<$ $\leq$, $>$, $\geq$), and k is a variable or numeric constant or, if the aggregation operator is `max` or `min`, a tuple of variables or numeric constants. Comparisons of the second form are first rewritten as two comparisons of the first form by introducing a fresh variable. For example, $agg\, S \neq agg'\, S'$ is rewritten as $agg\, S \neq V \wedge agg'\, S' = V$, and $agg\, S < agg'\, S'$ is rewritten as $agg\, S < V \wedge agg'\, S' \geq V$, where V is a fresh variable. The latter rewrite uses two inequalities, instead of an inequality and an equality, to increase the cases where occurrences of predicate atoms are positive (defined below).

We include a comprehensive set of comparison operators for readability; it also allows negation applied to comparisons to be eliminated by reversing the comparison operators; for example, the negation of a comparison using $<$ is a comparison using $\geq$.

The key idea here is that the value of a comparison (containing an aggregation) is undefined if there is not enough information about the predicates used to determine the value, or if applying the comparison (containing an aggregation) gives an error, such as a type error.

Additional aggregation and comparison functions, e.g., on other data types such as characters and strings, can be supported in the same principled way as we support those discussed here.

Programs, atoms, and literals. A *program* π is a set of rules and facts, plus declarations for predicates, described after dependencies are introduced next.

An *atom* of π is either a predicate symbol P in π applied to constants in the domain of π and variables, or a comparison formed using predicate symbols in π, constants in the domain of π, and variables. These are called *predicate atoms* for P and *comparison atoms*, respectively.

A *literal* of π is either an atom of π or the negation of a predicate atom of π. These are called *positive literals* and *negative literals*, respectively. A literal that is a predicate atom or its negation is called a *predicate literal*. A literal that is a comparison atom is called a *comparison literal*. Note that negation of a comparison atom is not needed because the negation will be eliminated by reversing the comparison operator.

Dependency graph. The dependency graph of a program characterizes dependencies between predicates induced by the rules, distinguishing positive from non-positive dependencies. We define the dependency graph before discussing declarations for predicates, because the permitted declarations and default declarations are determined by the dependency graph.

An occurrence A of a predicate atom in a hypothesis H is a *positive occurrence* if (1) H is A, which is a positive literal, (2) H is a quantification, and A is a positive literal in its body, (3) H is a comparison atom of the form `count` $S \geq k$, `count` $S > k$, `max` $S \geq k$, `max` $S > k$, `min` $S \leq k$, or `min` $S < k$, and A

is in a positive literal in the set expression S, or (4) H is a comparison atom of the form `count` $S \leq k$, `count` $S < k$, `max` $S \leq k$, `max` $S < k$, `min` $S \geq k$, or `min` $S > k$, and A is in a negative literal in the set expression S. A predicate atom that is the conclusion of a rule is also a positive occurrence. Other occurrences of predicate atoms are *non-positive occurrences*.

This definition conservatively ensures that hypotheses are monotonic with respect to positive occurrences of predicate atoms, i.e., making a positive occurrence of a predicate atom in a hypothesis true cannot make the hypothesis change from true. This definition can be extended so that any occurrence A of a predicate atom in a hypothesis H is a *positive occurrence* if H can be determined to be monotonic with respective to A. For example, if predicate `p` holds for only non-negative numbers, then `p(x)` is a positive occurrence in `sum {x: p(x)} > k`.

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 positive (respectively, non-positive) if a rule whose conclusion contains Q has a hypothesis that contains a positive (respectively, non-positive) occurrence of an atom for P. If there is a path from Q to P in $DG(\pi)$, then Q *depends on* P in π. If the node for P is in a cycle containing a non-positive edge in $DG(\pi)$, then P has *circular non-positive dependency* in π.

Declarations. A predicate declared *certain* means that each assertion of the predicate has a unique true (*True*) or false (*False*) value. A predicate declared *uncertain* means that each assertion of the predicate has a unique true, false, or undefined (*Undef*) value. A predicate declared *complete* means that all rules with that predicate in the conclusion are given in the program. A predicate declared *closed* means that an assertion of the predicate is set to false, called *self-false*, if inferring it to be true using the given rules and facts requires assuming itself to be true.

A predicate must be declared uncertain if it has circular non-positive dependency, or depends on 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. A predicate may be declared closed or not only if it is uncertain and complete, and it is by default not closed.

We do not give here a syntax for declarations of predicates to be certain, complete, closed, or not, because it is straightforward, and almost all examples use default declarations. However, Liu and Stoller [LS20b, LS21] introduces a language that supports such declarations and supports the use of both founded semantics and constraint semantics.

Notations. In presenting the semantics, in particular the completion rules, we allow negation in the conclusion of rules, and we allow hypotheses to be equalities ($=$) and negated equalities ($\neq$) between two variables or a variable and a constant.

4 Formal semantics

This section extends the definitions of founded semantics and constraint semantics in [LS18, LS20a] to handle aggregation and comparison. We introduce a new relation, namely, derivability of comparisons, and extend most of the foundational definitions, including the definitions of atom, literal, and positive occurrence in Section 3, and of complement, ground instance, truth value of a literal in an interpretation, completion rule, naming negation, unfounded set, and constraint model in this section. By carefully extending these foundational definitions, we are able to avoid explicit changes to the definitions of other terms and functions built on them, including the definition of completion function and the definition of the least fixed point at the heart of the semantics, embodied mainly in the function *LFPbySCC*.

4.1 Interpretations and derivability

Complements and consistency. The predicate literals A and $\neg A$ are *complements* of each other. The following pairs of comparison literals are complements of each other: $agg\,S = k$ and $agg\,S \neq k$; $agg\,S \leq k$ and $agg\,S > k$; $agg\,S \geq k$ and $agg\,S < k$.

A set of predicate literals is *consistent* if it does not contain a literal and its complement.

Ground instance. An occurrence of a variable X in a quantification Q is *bound* in Q if X is a variable to the left of the vertical bar in Q. An occurrence of a variable X in a set expression S is *bound* if X is a variable to the left of the colon in S. An occurrence of a variable in a rule R is *free* if it is not bound in a quantification or set expression in R.

A *ground atom* or *ground literal* is an atom or literal, respectively, not containing variables. A *ground instance* of a rule R in a program π is any rule obtained from R by expanding universal quantifications into conjunctions over all constants, instantiating existential quantifications with any constants, and instantiating the remaining free occurrences of variables with any constants (of course, all free occurrences of the same variable are replaced with the same constant). A *ground instance* of a comparison atom A is a comparison atom obtained from A by instantiating the free occurrences of variables in A with any constants. A *ground instance* of a set expression $\{X_1, ..., X_a : B\}$ is a pair $((X_1, ..., X_a), B)$ obtained by instantiating all variables in $X_1, ..., X_a$ and B with any constants.

Interpretations. An *interpretation* of a program π is a consistent set of ground predicate literals of π. Interpretations are generally 3-valued: a ground predicate literal is *true* (i.e., has truth value *True*) in interpretation I if it is in I, is *false* (i.e., has truth value *False*) in I if its complement is in I, and is *undefined* (i.e., has truth value *Undef*) in I if neither it nor its complement is in I. An interpretation of π is *2-valued* if it contains, for each ground predicate atom A of π, either A or its complement. Interpretations are ordered by set inclusion $\subseteq$.

Let $G(S)$ denote the set of ground instances of set expression S. For a set expression S, interpretation I, and truth value t, let

$$G(S, I, t) = \{x \mid (x, B) \in G(S) \wedge B \text{ has truth value } t \text{ in } I\}$$

That is, $G(S, I, t)$ is the set of combinations of constants for which the body of set expression S has truth value t in I.

Derivability of comparisons. Informally, a ground comparison atom $agg\,S \odot k$ is *derivable* in interpretation I of π, denoted $\pi, I \vdash agg\,S \odot k$, if the comparison must be true in I, regardless of whether atoms with truth value *Undef* are true or false.

Precisely, founded semantics uses the *linear-time derivability relation* $\vdash_{\mathrm{L}}$ defined in Figure 1 based on the aggregation operator and the comparison operator. It can be computed straightforwardly in linear time in $|G(S, I, \mathit{True})| + |G(S, I, \mathit{Undef})|$.

Derivability for each comparison in Figure 1 has also a condition that the comparison does not give an error. It gives an error if the aggregation gives an error, or if there is a type error, i.e., either the aggregation is `count` or `sum`, or is `max` or `min` with arity of S being 1, and k is not a number, or the aggregation is `max` or `min` with arity a of S greater than 1, and k is not an a-tuple of numbers. The aggregation gives an error if it is `max` or `min` and $G(S, I, \mathit{True}) \cup G(S, I, \mathit{Undef})$ is empty, or if there is a type error, i.e., either it is `max` or `min` and $G(S, I, \mathit{True})$ or $G(S, I, \mathit{Undef})$ contains either a non-number or a tuple containing a non-number, or it is `sum` and S has arity 1 and $G(S, I, \mathit{True})$ or $G(S, I, \mathit{Undef})$ contains a non-number, or it is `sum` and S

$$
\begin{array}{rcl}
\pi, I \vdash_{\mathrm{L}} \texttt{count}\, S = k & \Leftrightarrow & |G(S, I, \mathit{True})| = k \wedge G(S, I, \mathit{Undef}) = \emptyset \\
\pi, I \vdash_{\mathrm{L}} \texttt{count}\, S > k & \Leftrightarrow & |G(S, I, \mathit{True})| > k \\
\pi, I \vdash_{\mathrm{L}} \texttt{count}\, S < k & \Leftrightarrow & |G(S, I, \mathit{True}) \cup G(S, I, \mathit{Undef})| < k \\
\pi, I \vdash_{\mathrm{L}} \texttt{max}\, S = k & \Leftrightarrow & k \in G(S, I, \mathit{True}) \wedge \forall\, i \in G(S, I, \mathit{True}) \cup G(S, I, \mathit{Undef}) \mid i \leq k \\
\pi, I \vdash_{\mathrm{L}} \texttt{max}\, S \neq k & \Leftrightarrow & k \notin G(S, I, \mathit{True}) \cup G(S, I, \mathit{Undef}) \vee \exists\, i \in G(S, I, \mathit{True}) \mid i > k \\
\pi, I \vdash_{\mathrm{L}} \texttt{max}\, S > k & \Leftrightarrow & \exists\, i \in G(S, I, \mathit{True}) \mid i > k \\
\pi, I \vdash_{\mathrm{L}} \texttt{max}\, S < k & \Leftrightarrow & \exists\, i \in G(S, I, \mathit{True}) \wedge \forall\, i \in G(S, I, \mathit{True}) \cup G(S, I, \mathit{Undef}) \mid i < k \\
\pi, I \vdash_{\mathrm{L}} \texttt{sum}\, S = k & \Leftrightarrow & \text{sum}\ G(S, I, \mathit{True}) = k \wedge \{\text{first}(i) : i \in G(S, I, \mathit{Undef})\} \subseteq \{0\} \\
\pi, I \vdash_{\mathrm{L}} \texttt{sum}\, S > k & \Leftrightarrow & \text{sum}\ (G(S, I, \mathit{True}) \cup \{i \in G(S, I, \mathit{Undef}) : \text{first}(i) < 0\}) > k \\
\pi, I \vdash_{\mathrm{L}} \texttt{sum}\, S < k & \Leftrightarrow & \text{sum}\ (G(S, I, \mathit{True}) \cup \{i \in G(S, I, \mathit{Undef}) : \text{first}(i) > 0\}) < k
\end{array}
$$

Figure 1: Linear-time derivability relation for comparisons. first(i) returns the first component of i if i is a tuple, and returns i otherwise. Biconditionals ($\Leftrightarrow$) for derivability of other comparisons are obtained from those given as follows. (1) Biconditionals for deriving comparisons using `min` are obtained from those for `max` by replacing `max` with `min`, interchanging $\leq$ and $\geq$, and interchanging $<$ and $>$. (2) For aggregation operator *agg* being `count` or `sum`, the right side of the biconditional for deriving $\mathit{agg}\, S \neq k$ is the disjunction of the right sides of the biconditionals for deriving $\mathit{agg}\, S > k$ and $\mathit{agg}\, S < k$. (3) For each aggregation operator *agg*, biconditionals for deriving $\mathit{agg}\, S \geq k$ and $\mathit{agg}\, S \leq k$ are obtained from the given biconditionals for $\mathit{agg}\, S > k$ and $\mathit{agg}\, S < k$, respectively, by replacing $> k$ with $\geq k$ and replacing $< k$ with $\leq k$.

has arity greater than 1 and $G(S, I, \mathit{True})$ or $G(S, I, \mathit{Undef})$ contains a tuple whose first component is not a number. Comparisons that give errors can easily be detected and reported by checking these conditions.

This definition of derivability is relatively strict about errors, for example, it always makes a comparison give an error if the aggregation in it gives an error. One can be less strict about errors, for example, a comparison containing `max` or `min` applied to the empty set and using negated equality could be allowed to hold even if the aggregation in it gives an error, taking the view that an error is not equal to a value or a tuple of values in the domain. This generally yields more literals that are true or false, rather than undefined. Choices for error handling could also be specified using declarations.

An alternative to linear-time derivability is *exact derivability*, denoted $\vdash_{\mathrm{E}}$. Informally, $\pi, I \vdash_{\mathrm{E}} \mathit{agg}\, S \odot k$ holds iff (1) $\mathit{agg}\, S \odot k$ holds in all 2-valued interpretations I' that extend I and satisfy the part of π that S depends on, and (2) there is at least one such interpretation I'. Exact derivability is based on enumeration of interpretations and hence is less appropriate for founded semantics, which is designed to leave such enumeration for constraint semantics. Although exact derivability can be more precise in principle, linear-time derivability gives the same result as exact derivability for all examples we found in the literature.

Interpretations provide truth values for comparison literals similarly as for predicate literals. Let $DC(\pi, I)$ be the set of comparisons derivable for program π and interpretation I. A comparison literal A for π is *true* in I if it is in $DC(\pi, I)$, is *false* in I if its complement is in $DC(\pi, I)$, and is *undefined* in I otherwise.

Models. An interpretation I of a program π is a *model* of π if it (1) contains all facts in π, and (2) satisfies all rules of π, interpreted as formulas in 3-valued logic [Fit85] (i.e., for each ground instance of each rule, if the body is true in I, then so is the conclusion).

One-step derivability. The *one-step derivability* function T_π for program π performs one step of

inference using rules of π. Formally, $A \in T_\pi(I)$ iff (1) A is a fact of π, or (2) there is a ground instance R of a rule of π with conclusion A such that the body of R is true in interpretation I.

4.2 Founded semantics without closed declarations

We first define a version of founded semantics, denoted *Founded*$_0$, that ignores declarations that predicates are closed. We then extend the definition to handle those declarations. Intuitively, the *founded model* of a program π ignoring closed-predicate declarations, denoted *Founded*$_0(\pi)$, is the least set of literals that are given as facts or can be inferred by repeatedly applying the rules. Formally, we define

$$\mathit{Founded}_0(\pi) = \mathit{UnNameNeg}(\mathit{LFPbySCC}(\mathit{NameNeg}(\mathit{Cmpl}(\pi)))),$$

where functions *Cmpl*, *NameNeg*, *LFPbySCC*, and *UnNameNeg* are defined as follows.

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

The function *Combine*(π) 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. First, transform the facts and rules defining Q so they all have the same conclusion $Q(V_1, ..., V_a)$, by replacing each fact or rule $Q(X_1, ..., X_a) \leftarrow B$ with

$$Q(V_1, ..., V_a) \leftarrow (\exists\ Y_1, ..., Y_k \mid V_1 = X_1 \wedge\ ...\ \wedge V_a = X_a \wedge B)$$

where $V_1, ..., V_a$ are fresh variables (i.e., not occurring in any given rule defining Q), and $Y_1, ..., Y_k$ are all variables occurring free in the original rule $Q(X_1, ..., X_a) \leftarrow B$. Then, 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.

The function *AddInv*(π) 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 $A \leftarrow B$ is $\neg A \leftarrow \neg B$), by (1) 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, and (2) eliminating negation applied to comparison atoms by reversing the comparison operators. As a result, in completion rules, negation is applied only to predicate atoms.

Similar completion rules but without aggregation are used in Clark's completion [Cla78] and Fitting semantics [Fit85].

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, except where $P(X_1, ..., X_a)$ is a positive occurrence, $\neg P(X_1, ..., X_a)$ with $\texttt{n}.P(X_1, ..., X_a)$, and $P(X_1, ..., X_a)$ not after $\neg$ with $\neg\,\texttt{n}.P(X_1, ..., X_a)$. The new predicate $\texttt{n}.P$ represents the negation of predicate P. Since $P(X_1, ..., X_a)$ and $\neg P(X_1, ..., X_a)$ are complements of each other, we now also define $P(X_1, ..., X_a)$ and $\texttt{n}.P(X_1, ..., X_a)$ to be complements of each other.

Note that $\texttt{n}.P(X_1, ..., X_a)$ is introduced to make the one-step derivability function explicitly monotonic, while maintaining consistency. We replace $\neg P(X_1, ..., X_a)$ for any conclusion and any negative occurrence of $P(X_1, ..., X_a)$ in a hypothesis (where negative occurrence is defined symmetrically as positive occurrence) with $\texttt{n}.P(X_1, ..., X_a)$ to allow negative conclusions to be derived and used as facts. We replace any negative occurrence of $P(X_1, ..., X_a)$ not after $\neg$ with $\neg\,\texttt{n}.P(X_1, ..., X_a)$ also to use these

facts. Other occurrences of $\neg P(X_1, ..., X_a)$ and $P(X_1, ..., X_a)$ in hypotheses, if any due to positive (and negative) occurrence being conservative, can be either replaced or left, with the result still being a model, because all derivation and use of $\texttt{n}.P(X_1, ..., X_a)$ and $P(X_1, ..., X_a)$ follow the one-step derivability. We have not seen any example that needs the following, but one might obtain a more precise model, i.e., more atoms that are true or false, by trying all combinations of replacing and leaving. It is an open question whether some combination leads to a unique most precise model.

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 $C_1, ..., C_n$ be a list of the SCCs in dependency order, so predicates in earlier SCCs do not depend on predicates in later ones; it is easy to show that any linearization of the dependency order leads to the same result for *LFPbySCC*. The *projection* of a program π onto an SCC C, denoted *Proj*(π, C), contains all facts of π whose predicates are in C and all rules of π whose conclusions contain predicates in C.

Define $\mathit{LFPbySCC}(\pi) = I_n$, where $I_0 = \emptyset$ and $I_i = \mathit{AddNeg}(\mathit{LFP}(T_{\mathit{Proj}(\pi,C_i)\cup I_{i-1}}), C_i)$ for $i \in 1..n$. *LFP* is the least fixed point operator. $\mathit{AddNeg}(I, C)$ returns the interpretation obtained from interpretation I by adding *completion facts* for the certain predicates in C to I; specifically, for each certain predicate P in C, and each combination of values $v_1, ..., v_a$ of arguments of P, if I does not contain $P(v_1, ..., v_a)$, then add $\texttt{n}.P(v_1, ..., v_a)$.

The least fixed point is well-defined, because the one-step derivability function $T_{\mathit{Proj}(\pi,C_i)\cup I_{i-1}}$ is monotonic with respect to $\subseteq$, i.e., for all interpretations J and J', $T_{\mathit{Proj}(\pi,C_i)\cup I_{i-1}}(J) \subseteq T_{\mathit{Proj}(\pi,C_i)\cup I_{i-1}}(J')$ whenever $J \subseteq J'$. To establish this, we show monotonicity of each part of the definition of T_π. Part (1) adds a fixed set of facts and hence is trivially monotonic. Part (2) is monotonic because derivability of comparisons is monotonic with respect to $\subseteq$ (i.e., $DC(\pi, J) \subseteq DC(\pi, J')$ whenever $J \subseteq J'$), and because every rule that can "fire" in J (i.e., its hypotheses are true) can also fire in J'.

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

4.3 Founded semantics with closed declarations

Informally, when an uncertain complete predicate is declared closed, an atom A of the predicate is false in an interpretation I for a program π, called *self-false* in I, if every ground instance of a rule that concludes A has a hypothesis that is false in I or, recursively, is self-false in I. To simplify the formalization, we first transform ground instances of rules to eliminate disjunction, by putting the body of each ground instance R of a rule into disjunctive normal form (DNF) and then replacing R with multiple rules, one per disjunct of the DNF.

A set U of ground predicate atoms for closed predicates is an *unfounded set* of π with respect to an interpretation I of π iff U is disjoint from I and, for each atom A in U, and each ground instance R of a rule of π with conclusion A,

(1) some hypothesis of R is false in I,

(2) some positive predicate hypothesis of R is in U, or

(3) some comparison hypothesis H of R is false when all atoms in U are false, i.e., $\pi, I \cup \neg \cdot U \vdash_{\mathrm{L}} \neg H$,

where, for a set S of positive literals, $\neg \cdot S = \{\neg P(c_1, ..., c_a) \mid P(c_1, ..., c_a) \in S\}$, called the *element-wise negation* of S, and where $\neg H$ is implicitly simplified to eliminate negation applied to H by changing the comparison operator in H.

Note that this definition differs from the standard definition of unfounded set [VRS91] in that we restricted the unfounded set to atoms for closed predicates, added clause (3), and added the disjointness

condition. Because a comparison hypothesis depends non-conjunctively on the truth values of multiple literals for predicates used in the aggregation, and these literals may be spread across I and U, clause (3) checks whether H is false when all atoms in U are set to false. The explicit disjointness condition is not needed in WFS or founded semantics without aggregation, because one can prove in those settings that unfounded sets are disjoint from interpretations that arise in the semantics (e.g., see [VRS91, Lemma 3.4]). The disjointness condition is needed here to ensure that the interpretation $I \cup \neg \cdot U$ in clause (3) is consistent and hence the meaning of the clause is well-defined.

The definition of unfounded set U ensures that extending I to make all atoms in U false is consistent with π, in the sense that no atom in U can be inferred to be true in the extended interpretation. We define $\mathit{SelfFalse}_\pi(I)$, the set of *self-false atoms* of π with respect to interpretation I, to be the greatest unfounded set of π with respect to I. Note that this set is empty when no predicate is declared closed.

The founded semantics is defined by repeatedly computing the semantics given by $\mathit{Founded}_0$ (founded semantics without closed declarations) and then setting self-false atoms to false, until a least fixed point is reached. Formally, the founded semantics is $\mathit{Founded}(\pi) = \mathit{LFP}(F_\pi)$, where $F_\pi(I) = \mathit{Founded}_0(\pi \cup I) \cup \neg \cdot \mathit{SelfFalse}_\pi(\mathit{Founded}_0(\pi \cup I))$.

4.4 Constraint semantics

Constraint semantics is a set of 2-valued models based on founded semantics. A *constraint model* M of a program π is a 2-valued interpretation of π such that (1) $\mathit{Founded}(\pi) \subseteq M$, (2) M is a model of $\mathit{Cmpl}(\pi)$, and (3) if there are closed predicates, there is no non-empty subset S of $M \setminus \mathit{Founded}(\pi)$ such that S contains only positive literals for closed predicates and $S = \mathit{SelfFalse}_\pi(M \setminus S)$. Condition (3) says that M should not contain a set S of positive literals for closed predicates that are not required to be true by the founded semantics and are self-false with respect to the rest of M.

We also require that an interpretation that leads to an error in a comparison be not a constraint model. Precisely, we require that for interpretation M to be a constraint model, no ground instance of a rule of π contains a comparison that gives an error in M. Errors are defined the same as in Section 4.1, but note that $G(S, I, \mathit{Undef})$ is empty here. This definition of constraint models could also be made less strict about errors as in Section 4.1.

Note that condition (3) differs from the corresponding condition in constraint semantics without aggregation [LS18, LS20a], which is $\neg \cdot \mathit{SelfFalse}(M) \subseteq M$. The change is needed because of the new disjointness condition for unfounded sets. With the new disjointness condition, for any 2-valued interpretation M, $\mathit{SelfFalse}(M)$ must be empty, and hence $\neg \cdot \mathit{SelfFalse}(M) \subseteq M$ is vacuously true.

We define $\mathit{Constraint}(\pi)$ to be the set of constraint models of π. Constraint models can be computed by iterating over interpretations M that are supersets of $\mathit{Founded}(\pi)$, thus satisfying condition (1), and then checking whether conditions (2) and (3) are satisfied.

4.5 Range-blocked inference

Use of `sum` or `count` can generate values outside the representable range $[-NRB, NRB]$. Our semantics only considers predicate argument values in the domain, and thus does not infer facts with an argument value outside the representable range. We call inferences that would derive such facts "range-blocked inferences".

Formally, program π has a *range-blocked inference* in interpretation I if there is a rule R of π such that: (1) R has a hypothesis of the form $\mathit{agg}\, S = k$, (2) k is a variable that also occurs in the conclusion of R, and (3) there is a ground instance R' of R, containing a ground instance $\mathit{agg}\, S' = k'$ of $\mathit{agg}\, S = k$, such that (i) all hypotheses of R' other than $\mathit{agg}\, S' = k'$ are true in I and (ii) either $\pi, I \models \mathit{agg}\, S' > NRB$

or $\pi, I \models agg\, S' < -NRB$. We can detect and report range-blocked inferences by checking the conditions in this definition.

This definition is designed to be permissive and allows inferences involving intermediate values (i.e., values not in the inferred facts) outside the representable range. For example, it allows inference using the rule `q` $\leftarrow$ `sum {x: p(x)} > 0` even when the sum is outside the representable range. The definition can easily be extended to block such inferences as well; this may better reflect implementation behavior, because implementations need to compute intermediate values.

5 Properties of the semantics

Consistency and correctness.
Theorem 1. The founded model and constraint models of a program π are consistent.
Proof. See Appendix A. ■

Theorem 2. The founded model of a program π is a model of π and $Cmpl(\pi)$. The constraint models of π are 2-valued models of π and $Cmpl(\pi)$.
Proof. See Appendix A. ■

Same SCC, same certainty. All predicates in an SCC have the same certainty. The proof is the same as for Theorem 4 in [LS20a].
Theorem 3. For every program, for every SCC C in its dependence graph, all predicates in C are certain, or all of them are uncertain.

Equivalent declarations. Changing predicate declarations from uncertain, complete, and closed to certain when allowed, or vice versa, preserves founded and constraint semantics. The same SCC, same certainty property above implies that this change needs to be made for all predicates in an SCC. This property is formally stated in the following theorem. Several modifications to the proof of the corresponding theorem in [LS20a] are needed.

When predicates are declared certain, founded semantics and constraint semantics are the same and can be computed much more efficiently than the worst case, by simple least fixed-point computations, without using completion rules, finding self-false atoms, or doing constraint solving.
Theorem 4. Let π be a program. Let C be an SCC in its dependency graph containing only predicates that are uncertain, complete, and closed. Let π' be a program identical to π except that all predicates in C are declared certain. Note that, for the declarations in both programs to be allowed, predicates in all SCCs that follow C in dependency order must be uncertain, predicates in all SCCs that precede C in dependency order must be certain, and predicates in C must not have circular non-positive dependency. Then $Founded(\pi) = Founded(\pi')$ and $Constraint(\pi) = Constraint(\pi')$.
Proof. See Appendix A. ■

Higher-order programming. Founded semantics and constraint semantics are preserved by a transformation that facilitates higher-order programming by replacing a set S of compatible predicates with a single predicate `holds` whose first argument is the name of one of those predicates. For example, if `win` is in S, then `win(x)` is replaced with `holds('win',x)`. The definitions of compatible predicates and of the program transformation that replaces predicates in S with `holds` are the same as in [LS20a], with the clarification that the auxiliary function $MergeAtom_S(A)$, which replaces predicates in S with `holds` in an atom A, is applied only to predicate atoms. The proof is the same as for Theorem 5 in [LS20a].

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 Discussion

This section discusses and compares with major prior semantics for recursive rules with aggregation. In these comparisons, including the given results for examples, default declarations for the predicates are used in our semantics except where stated otherwise. In particular, our semantics is able to match all desired results from prior semantics.

Stratified and monotonic programs. Many prior works, especially in deductive databases, consider only stratified programs, which prohibit recursion through aggregation or negation, or monotonic programs, e.g., [MPR90, RS92, SYZ15, YSZ17, ZYD+17, ZDG+19]. These works vary in the details of the rule language and the programs recognized as stratified or monotonic. In our semantics, all these programs fall in the class of programs where all predicates can be declared certain, and whose semantics are fully determined by least fixed-point computations.

First-order logic. Several prior works extend first-order logic with various kinds of aggregations. For example, Hella et al. [HLNW99, HLNW01] extend first-order logic with aggregation operators similar to those in database query languages, and analyze its expressiveness. Programs in our language can be regarded as sets of formulas in such logics, with $\leftarrow$ representing reverse implication, and with aggregations translated to accommodate syntactic differences between the languages. If all predicates in the program are uncertain and not complete, constraint semantics of the program matches the semantics of those formulas, i.e., the constraint models of the program are exactly the models of those formulas. Theorem 7 in [LS20a] establishes this result without aggregations. It is easy to see that the result still holds when aggregations are considered, because our semantics for aggregations in 2-valued models matches the standard semantics for aggregations used in such logics.

Kemp-Stuckey 1991. Kemp and Stuckey [KS91] is one of the earliest comprehensive studies, improving over a number of previous works that handled limited classes of programs with aggregations. They extend WFS and SMS to logic programs with aggregations and study previously defined classes of programs with aggregations under several notions of stratification as well as properties such as monotonicity.

Their extension of WFS requires that a set be fully defined before aggregation can be evaluated on it; thus, if the predicate used in an aggregation is *Undef* for any argument, the truth value of the hypothesis containing that aggregation is also *Undef*. This is much less precise than founded semantics, in the sense that it leaves many more atoms with truth value *Undef*. For example, for the company control problem [MPR90], they point out that their extension of WFS does not infer the expected results, while founded semantics and constraint semantics do, as shown in Section 7.5.

Another way in which founded semantics is more precise is that their extension of WFS uses the standard definition of unfounded set, while our semantics extends that definition to handle aggregation precisely, allowing larger unfounded sets and thus more self-false atoms, yielding fewer undefined values. For example, for the correlated counts problem discussed in Section 7.4, if `p` is declared uncertain, complete, and closed, instead of the default certain, this additional clause is needed to conclude that {`p(2)`, `p(3)`} is an unfounded set.

In extending SMS to handle aggregations, they treat literals containing aggregation as negative literals when computing the program reduct. However, they found that this may produce un-intuitive non-minimal stable models. For example, for the company control problem, it allows undesired stable models [KS91, Section 6], while founded semantics and constraint semantics yield exactly the desired model, as shown in Section 7.5.

Van Gelder 1992. Van Gelder [VG92] presents an early approach in which aggregations are defined using ordinary rules, rather than introduced as new primitives, in a language with 3-valued semantics. It illustrates the approach using examples involving min, max, subset, and sum, with the rules defining the aggregations customized in some cases to the problem at hand, with the comment "*ad hoc* methods of analysis seem to be necessary."

The paper shows that the desired results are obtained for several non-trivial examples but not for some others. For example, for the company control problem in Section 7.5, Van Gelder points out that his semantics sometimes leaves atoms for the `controls` predicate undefined, even though they can be determined to be true or false using the given `ownsStk` facts [VG92, Example 6.1]; our semantics infers all of the expected `controls` relations.

Unfortunately, it is hard to characterize the programs for which their approach gives desired results because of their use of ad hoc methods. In contrast, our work handles a clearly defined language of programs with aggregations, allows specification of different assumptions, and supports both 2-valued and 3-valued semantics. Also, our work allows rules with disjunction and quantifiers. These are not considered in [VG92].

Pelov et al. 2007. Pelov, Denecker, and Bruynooghe [PDB07] present a framework in which semantics of logic programs are obtained by approximating the fixed points of an immediate consequence operator. They define the *ultimate approximation* U^{aggr} of this operator extended to handle aggregations, and define the ultimate Fitting (Kripke-Kleene), ultimate WFS, and ultimate SMS of programs with aggregations as appropriate kinds of fixed points of U^{aggr}.

Their ultimate semantics are intended to be most precise but have very high computational complexity. Therefore, Pelov et al. also introduce more tractable semantics. They define extensions Φ^{agg} of Fitting's 3-valued approximation of the immediate consequence operator Φ to handle aggregations, and obtain extensions of Fitting semantics, WFS, and SMS as appropriate kinds of fixed points of Φ^{agg}. They consider two versions of Φ^{agg}, based on "trivial" and "ultimate" 3-valued approximations of the semantics of aggregations.

To see the precision of their ultimate Fitting semantics and ultimate WFS, consider the program

```
p(1)   p(-1)
q ← ¬ q
c ← sum {x: p(x) ∧ q} ≥ 0
```

In both their ultimate Fitting semantics and ultimate WFS, `c` is undefined. To see this, first note that `q` is undefined, so possible values of the set being summed range from $\emptyset$ (in interpretations where `q` is false) to $\{\texttt{-1}, \texttt{1}\}$ (in interpretations where `q` is true). Their framework uses intervals in the subset lattice, so the comparison in the third rule is evaluated for all sets in the interval from $\emptyset$ to $\{\texttt{-1}, \texttt{1}\}$ in the subset lattice, and the comparison is true for some of them (e.g., $\emptyset$) and false for others ($\{\texttt{-1}\}$), so `c` is left undefined. Founded semantics matches that result, because the comparison in the third rule is not derivable. Founded semantics using exact derivability would give a more precise result: the comparison is evaluated only for $\emptyset$ and $\{\texttt{-1}, \texttt{1}\}$ as possible values of the set expression, and is true for both, so `c` is true, as one would expect for a most precise semantics.

To explore assumptions underlying their extension of SMS, consider the following program used by Faber et al. [FPL11, Example 5.2] to compare with other semantics including Pelov et al.'s:

```
p(1) ← ¬ p(-1)
p(-1) ← p(1)
p(1) ← sum {x: p(x)} ≥ 0
```

In Faber et al.'s semantics, this program has one model, {p(1), p(-1)}. Our constraint semantics (with default declarations) matches Faber et al.'s semantics. In Pelov et al.'s semantics, this program has no stable models. Our constraint semantics, when p is declared closed (which is not the default), matches Pelov et al.'s semantics.

Faber et al. 2011. Faber, Pfeifer, and Leone [FPL11] define an answer set semantics (analogous to SMS) for disjunctive programs with aggregations, using a generalization of the Gelfond-Lifschitz transformation. They also analyze the computational complexity of the language for several different restrictions on the allowed aggregations.

To explain their semantics that differs from other prior semantics, consider the following two programs used by Gelfond and Zhang [GZ19, Example 2] to help motivate a different semantics. The first program, called P_2, is

```
p(1) ← count {x: p(x))} ≥ 0
```

The second, called P_3, is a variant of P_2 logically expected to have the same meaning:

```
p(1) ← count {x: p(x))} = y, y ≥ 0
```

In Faber et al.'s semantics, P_2 has one model, {p(1)}, as in semantics for simple monotonic programs. Our semantics matches this: p is certain by default, because it has only a positive dependence on itself, and the founded semantics and constraint semantics are the same, giving the same one model.

In Faber et al.'s semantics, P_3 has no model, which may be surprising. Founded semantics provides an explanation: p is uncertain and complete by default, and p(1) is undefined. Constraint semantics gives the same one model as for P_2, matching the logically expected equivalence with P_2.

Gelfond-Zhang 2019. Gelfond and Zhang [Gel02, GZ14, ZR16, GZ17, GZ18, GZ19] study the challenges and solutions for aggregation in recursion extensively, in an effort to establish the desired semantics for aggregation that corresponds to SMS, a set of 2-valued models. This resulted in changes from earlier semantics by Gelfond [Gel02], essentially to capture an implicit closed-world assumption. Their most recent work [GZ19] introduces a new semantics, with programs P_2 and P_3 above as motivating examples.

They note that many prior semantics unintuitively give different results for programs P_2 and P_3 above. They resolve this problem by introducing a semantics in which both programs [GZ19, Example 3] have no model. Having no model for P_2 contradicts all other semantics we have seen that support aggregation in recursion, including semantics for even monotonic programs supported in deductive databases. To make P_2 and P_3 logically equivalent, the desired way is to make P_3 have the same model {p(1)} as P_2. Our constraint semantics achieves that, as described above.

Precision, range, and errors. Computing aggregations require operations on numbers, which incur the issues with precision, bound, and errors with numeric computations in reality. These issues are not explicitly addressed systematically, or not discussed at all, in prior work.

We describe the choices and the handling of these issues in our language and semantics: precision and range under Domain in Section 3, Range-blocked inference in Section 4.5, and errors under Derivability of comparisons in Section 4.1 and interpretations of comparisons in Constraint semantics in Section 4.4.

In general, choices for handling precision, range, errors could also be specified using declarations. We do not explicitly name these choices, because they are well-known in numeric computations in practice, and they are orthogonal to the declarations about predicates in our language and semantics.

7 Additional examples and applications

Many small examples similar to the example in Section 2 have been discussed extensively in the literature. The most recent work [GZ19] is most comprehensive in discussing 28 examples; we discuss their Examples 1 and 28 to show the range of difficulties they deal with, Example 15 that resorts to a subset relation, and Example 25 that spans the most discussion. We then discuss the well-known challenging company control problem [MPR90] and two even more challenging game problems that generalize the well-known win-not-win game [LS18, LS20a]. In all cases, founded semantics and constraint semantics are simple and give the desired results.

7.1 Classes needing teaching assistants

This is Example 1 in [GZ19]. It considers a complete list of students enrolled in a class `c`, represented by a collection of facts:

```
enrolled('c','mike')  enrolled('c','john')  ...
```

It defines a relation `need_ta(c)` that holds iff class `c` needs a teaching assistant, i.e., the number of students enrolled in the class is greater than 20, and it gives a second rule for its negation, as follows:

```
need_ta(c) ← count {x: enrolled(c,x)} > 20
n_need_ta(c) ← ¬ need_ta(c)
```

Because `enrolled` is certain from the list being complete, and there is no aggregation in recursion, `need_ta` is certain by default, and so is `n_need_ta`. Thus founded semantics and constraint semantics are the same and are straightforward to compute. First, `need_ta` is inferred by just doing the counting for each `c` and then checking if the count is greater than 20. Then `n_need_ta` is computed, simply concluding true for classes for which `need_ta` is false.

7.2 Graduation requirements—directly using universal quantification

This is Example 15 in [GZ19]. It considers a knowledge base containing two complete lists of facts, for two relations `taken` and `required`:

```
taken('mike','cs1')  taken('mike','cs2')
taken('john','cs2')
required('cs1')  required('cs2')
```

It introduces a subset relation to define a new relation `ready_to_graduate(s)` that holds if student `s` has taken all the required classes:

```
ready_to_graduate(s) ← {c: required(c)} ⊆ {c: taken(s,c)}
```

The problem description in [GZ19, Example 15] says that using the subset relation "avoids a more complex problem of introducing universal quantifiers and some kind of implication in the rules of the language".

With our language, the rule can be written directly using universal quantification and implication, where `P`$\rightarrow$`Q` can be trivially rewritten as $\neg$`P`$\vee$`Q`, yielding:

```
ready_to_graduate(s) ← ∀ c | ¬ required(c) ∨ taken(s,c)
```

Because `taken` and `required` as given are certain, and there is no negation or aggregation in recursion, `ready_to_graduate` is certain by default and can be computed simply as a least fixed point. This yields the same result for founded semantics and constraint semantics: `ready_to_graduate('mike')` is true and `ready_to_graduate('john')` is false.

Example 15 in [GZ19] also discusses other assumptions and rules. They are either non-issues or straightforward to handle in our language. For example, if `taken` is not complete, founded semantics gives that `ready_to_graduate('mike')` is true and `ready_to_graduate('john')` is undefined, and constraint semantics gives two models: one with `ready_to_graduate('john')` being true and one with it being false.

7.3 Digital circuits—from the most complex to the simplest

This is Example 25 in [GZ19], finishing the longest span of discussion by building on their Examples 11, 23, and 24 that are simpler instances or parts. It considers a program for propagating binary signals through a digital circuit that does not have a feedback, consisting of the following facts (where `input(w,g)` means that wire `w` is an input to gate `g`, `output(w,g)` is similar, `gate(g,'and')` means that gate `g` is an `and` gate, and `val(w,v)` means that wire `w` has value `v`):

```
input('w1','g1')  input('w2','g1')  input('w0','g2')
output('w0','g1')  output('w3','g2')
gate('g1','and')  gate('g2','and')
val('w1',0)  val('w2',1)
```

and a rule:

```
val(w,0) ← output(w,g) ∧ gate(g,'and')
           ∧ count {w: val(w,0), input(w,g)} > 0
```

Their Example 11 does not have the last fact on each line (i.e., no gate `g2`, input on `w0`, output on `w3`, and value on `w2`).[1] Their Examples 23 and 24 use simpler instances of the rule to illustrate their definitions of "splitting set" and "stratification", respectively.

First, `input`, `output`, and `gate` as given are certain. Then, `val` is certain by default, despite that `val` is defined using `val` in aggregation, because the dependency is positive—counting with $>$ and with no negation is monotonic. Therefore, the semantics is simply a least fixed point by using the given rule, yielding the same result for founded semantics and constraint semantics: `val('w0',0)`, `val('w3',0)`, plus the given facts, consistent with all of Examples 11 and 23–25 in [GZ19].

7.4 Correlated counts—with different predicate declarations

This is Example 28, the last example, in [GZ19]. It considers the following one fact and two rules:

```
p(1)
p(3) ← count {x: p(x)} ≥ 2
p(2) ← count {x: p(x)} ≥ 2
```

[1] Their Example 11 also reverses the first two hypotheses of the rule; this appears to be accidental.

We have that p is certain by default, despite that p is defined using p in aggregation, because the dependency is positive—counting with $\geq$ and with no negation is monotonic. The least fixed point infers p(1) being true in one iteration, and ends with p(1) being true, and p(3) and p(2) being false, as the result of both founded semantics and constraint semantics. This is the same as the resulting answer set in [GZ19], but is obtained straightforwardly, not using all possible guesses followed by sophisticated reducts as in computing answer sets, which, for this example and for one of the answer sets, considers 9 S-reducts, each containing 3 rules or clauses for a combination of three models each containing 2 of 3 possible facts [GZ19].

We also consider results under other assumptions, not discussed in [GZ19]. Suppose that the default is not used, and p is declared uncertain and complete. Then the following completion rule is added, and it does not infer p(3) or p(2) to be false.

```
¬ p(x) ← x ≠ 1 ∧ (x ≠ 3 ∨ count {x: p(x)} < 2)
               ∧ (x ≠ 2 ∨ count {x: p(x)} < 2)
```

Founded semantics gives that p(1) is true, and p(3) and p(2) are undefined. Constraint semantics gives two models: {p(1)} and {p(1),p(2),p(3)}.

Suppose that p is declared uncertain and not complete. It is straightforward that p(1) is true as given, and p(3) and p(2) are left as undefined. Thus, founded semantics and constraint semantics are the same as when p is uncertain and complete.

Supposed that p is declared uncertain, complete, and closed. Then the greatest unfounded set is {p(3),p(2)}, and founded semantics gives that p(1) is true, and p(3) and p(2) are false. That is, it makes the last two false, instead of undefined, and is the same as when p is certain. Since there are no undefined values, constraint semantics has one model: {p(1)}. This is again the same as in [GZ19], but note that using p being certain as above yields this desired result straightforwardly.

7.5 Company control—a well-known challenge

This is Examples 1.1 and 2.13 in [FPL11] and is also used in Example 12 in [GZ19]. The problem was also discussed repeatedly before, e.g., [MPR90, KS91, VG92, RS97, PDB07], and earlier [CGT90]. It considers a set of facts of the form company(c), denoting that c is a company, and a set of facts of the form ownsStk(c1,c2,p), denoting the percentage p of shares of company c2 that are owned by company c1. It defines that company c1 controls company c2, denoted controls(c1,c2), if the sum of the percentages of shares of c2 that are owned either directly by c1 or by companies controlled by c1 is more than 50.

```
controlsStk(c1,c1,c2,p) ← ownsStk(c1,c2,p)
controlsStk(c1,c2,c3,p) ← company(c1)
          ∧ controls(c1,c2) ∧ ownsStk(c2,c3,p)
controls(c1,c3) ← company(c1) ∧ company(c3)
          ∧ sum {p,c2: controlsStk(c1,c2,c3,p)} > 50
```

It introduces controlsStk(c1,c2,c3,p), denoting that company c1 controls p percent of shares of company c3 through company c2. It has become a most well-known challenging example for recursion with aggregation, because it involves aggregation in mutual recursion.

Founded semantics and constraint semantics are again straightforward to compute. First, company and ownsStk as given are certain. Then, controlsStk and controls are certain by default, despite that controlsStk and controls are mutually recursive while involving aggregation, because controlsStk(c1,c2,c3,p) holds for only non-negative p, making the dependency through the comparison positive. Therefore, the semantics is simply a least fixed point using the given rules, giving the same result for founded semantics and constraint semantics. This is the desired result, same as in [FPL11].

7.6 Double-win game—for any kind of moves

Consider the following game, which we call the double-win game. Given a set of moves, the game uses the following single rule, called double-win rule, for winning:

```
dwin(x) ← count {y: move(x,y) ∧ ¬ dwin(y)} ≥ 2
```

It says that `x` is a winning position if the number of positions, `y`, such that there is a move from `x` to `y` and `y` is not a winning position, is at least two. That is, `x` is a winning position if there are at least two positions to move to from `x` that are not winning positions.

We created the double-win game by generalizing the well-known win-not-win game [LS18, LS20a], which has a single rule, stating that `x` is a winning position if there is a move from `x` to some position `y` and `y` is not a winning position:

```
win(x) ← move(x,y) ∧ ¬ win(y)
```

One could also rewrite the double-win rule using two explicit positions `y1` and `y2` and adding `y1!=y2`, but this approach does not scale when the count can be compared with any number, not just 2, and is not necessarily known in advance.

By default, `move` is certain, and `dwin` is uncertain but complete. First, add the completion rule:

```
¬ dwin(x) ← count {y: move(x,y) ∧ ¬ dwin(y)} < 2
```

Then, rename ¬ `dwin` to `n.dwin`, in both the given rule and the completion rule, except the positive occurrence of `dwin` in the body of the completion rule, yielding:

```
dwin(x) ← count {y: move(x,y) ∧ n.dwin(y)} ≥ 2
n.dwin(x) ← count {y: move(x,y) ∧ ¬ dwin(y)} < 2
```

Now compute the least fixed point. Start with the base case, in the second rule, for positions `x` that have moves to fewer than 2 positions; this infers `n.dwin(x)` facts for those positions `x`. Then, the first rule infers `dwin(x)` facts for any position `x` that can move to 2 or more positions for which `n.dwin` is true.

This process iterates to infer more `n.dwin` and more `dwin` facts, until a fixed point is reached, where `dwin` gives winning positions, `n.dwin` gives losing positions, and the remaining positions are draw positions, corresponding to positions for which `dwin` is true, false, and undefined, respectively.

7.7 Over-win game—winning defined over winning

Consider the following game, which we call the over-win game. Given a set of moves, the game uses the following single rule, called the over-win rule, for winning:

```
owin(x) ← count {y: move(x,y) ∧ owin(y)} ≤ 2
```

It says that `x` is a winning position if the number of positions, `y`, such that there is a move from `x` to `y` and `y` is a winning position, is at most two. That is, `x` is a winning position if there are at most two positions to move to from `x` that are winning positions.

We created this over-win game from double-win game by removing the negation on the recursive occurrence of the predicate and reversing the comparison ≥ to ≤. That is, winning is now defined over winning, so to speak, instead of over not winning. In general, one could of course use any integer in place of 2. The point is, even though the recursive occurrence of the predicate is no longer negated, the occurrence of the predicate is still non-positive because of the reversed comparison.

By default, `move` is certain, and `owin` is uncertain but complete. First, add the completion rule:

```
¬ owin(x) ← count {y: move(x,y) ∧ owin(y)} > 2
```

Then, in the given rule, rename the non-positive occurrence of `owin` to `¬ n.owin`, and in the completion rule, rename `¬ owin` to `n.owin`, yielding:

```
owin(x) ← count {y: move(x,y) ∧ ¬ n.owin(y)} ≤ 2
n.owin(x) ← count {y: move(x,y) ∧ owin(y)} > 2
```

Now compute the least fixed point. Start with the base case, in the first rule, for positions `x` that have moves to no more than 2 positions; this infers `n.owin(x)` facts for those positions `x`. Then, the second rule infers `n.owin(x)` facts for any position `x` that can move to more than 2 positions for which `owin` is true.

This process iterates to infer more `owin` and more `n.owin` facts, until a fixed point is reached, where `owin` gives winning positions, `n.owin` gives losing positions, and the remaining positions are draw positions, corresponding to positions for which `owin` is true, false, and undefined, respectively.

8 Experiments

We also performed experiments with our new semantics and compared with results computed by four well-known systems. We implemented straightforward and incremental least fixed-point computations for selected examples in DistAlgo [LSL17], an extension of Python.

For comparison with founded semantics, we use XSB [SWS+17], the most well-known system for computing WFS and that supports negation and aggregation in recursion, and SWI-Prolog [WSTL12], which added similar support more recently. For comparison with constraint semantics, we use clingo [AFG15] and DLV [FPL+08, ACD+17], the most well-known systems for computing SMS and that support negation and aggregation in recursion.

We show experimental results for the three most complex examples we discussed: the company-control problem, the double-win game, and the over-win game. For our programs, the default declarations described in Section 7 are used. The XSB programs were written by an XSB expert. The SWI-Prolog programs are the same except for using a conjunction with `fail` to find all solutions in place of `do_all` that is in XSB but not in SWI-Prolog. For the company control problem, the DLV and clingo programs are the same, from [FPL11, Example 1.1]. For all benchmarks, we compared results from our system with results from the other logic rule engines to check correctness. We also manually checked correctness of the results for all small benchmarks.

While the emphasis of our experiments is on semantics and correctness, we also report running time results for the company-control problem. For the double-win game and over-win game, all other systems either do not compute the desired semantics or were found to compute incorrect results. XSB and SWI-Prolog have since found and fixed bugs that caused the incorrect results in the double-win game, but developers of these systems also found that fundamentally different inference would be needed to compute correctly at all for the over-win game.

Our experiments used five systems: DistAlgo 1.1.0b15, available via `https://github.com/DistAlgo/distalgo`, on Python 3.7.0; XSB 4.0.0, available via `http://xsb.sourceforge.net/`; SWI-Prolog 8.5.1, available via `https://www.swi-prolog.org/`; clingo 5.4.0, available via `https://potassco.org/clingo/`; and the version of DLV available via `https://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/` (accessed 2020-09-21 and 2021-10-01). That version of DLV supports recursive aggregates, while the current release of DLV "does not yet contain a full implementation of recursive aggregates" according to the DLV System website, `http://www.dlvsystem.com/dlv/` (accessed 2020-09-21 and 2021-10-01). We tried to run the company control problem in the current version of DLV (accessed

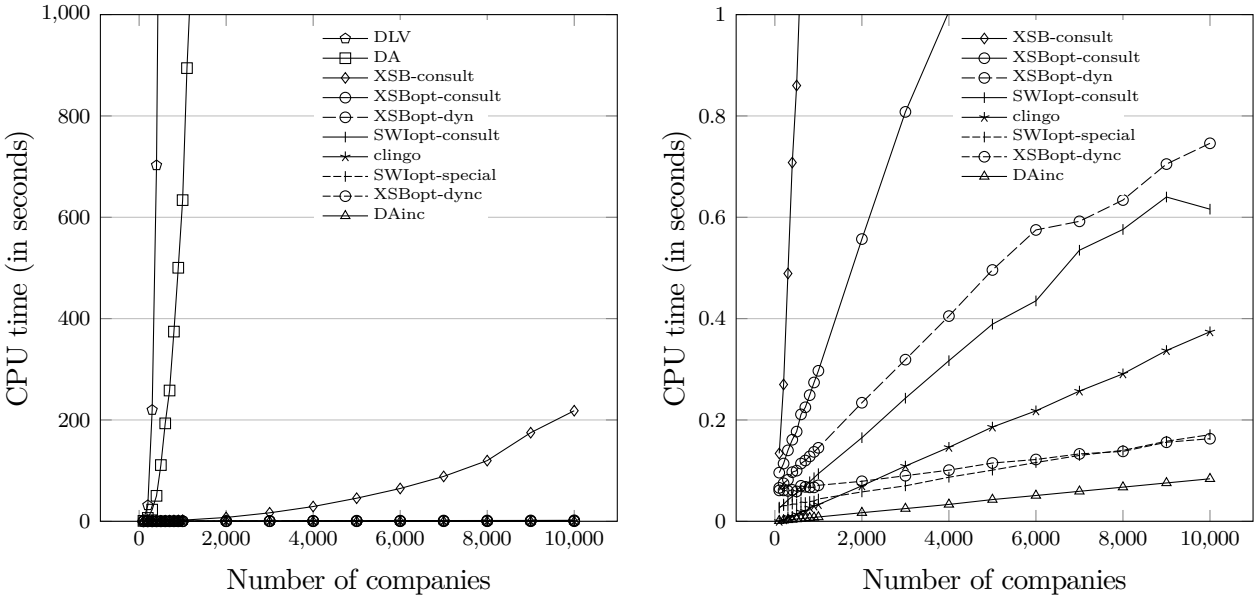

Figure 2: Running time comparison for the company control problem, showing CPU times up to 1000 seconds (left) and zooming in to 1 second to reveal the differences between the fastest systems (right). DA and DAinc are the DistAlgo least fixed point computation and the incrementalized version of it, respectively. XSBopt and SWIopt are XSB and SWI-Prolog, respectively, on the variant with reversed rules. For XSB and SWI-Prolog, the suffixes after the dash indicate how facts were loaded.

2021-10-01), but it exited with the error message "the predicates appearing in the aggregate atom cannot be recursive with the head of the rule".

Company control. For the company control problem, we compare a straightforward least fixed-point computation of our semantics in DistAlgo, and a more efficient version of that computation obtained using the incrementalization method developed by Liu et al. [LS09, LBSL16, LSL17], with XSB, SWI-Prolog, clingo, and DLV.

Our benchmark problem instance generator takes the desired number N of `company` facts as input and generates pseudorandom problem instances with N `company` facts and N `ownsStk` facts. The generator ensures that the owned percentages of each company sum to at most 100, and that nontrivial transitive control relationships exist. It accomplishes the latter by pseudorandomly choosing triples `c1`, `c2`, `c3` of companies, giving `c1` a controlling share (i.e., more than 50%) of `c2`, and splitting a controlling share of `c3` between `c1` and `c2`.

All measurements were taken on 64-bit Ubuntu 16.04 on a 3.47 GHz Intel Xeon X5690 CPU with 94 GB RAM. All running times are for computing all inferred facts as the semantics of the rules and include the time needed to read the `company` and `ownsStk` facts from a file and print the `controls` relation. Each reported time is an average of CPU times of runs over 10 generated benchmarks of the specified size. Our test driver checks and confirms that all five systems produce the same query answers for each benchmark.

Figure 2 shows the running times of the five systems for an increasing number of companies. For XSB and SWI-Prolog, besides the original rules, we also used an optimized variant with "reversed" rules; specifically, the `company` hypotheses in the second and third rules are moved to the ends of the rules.

This variant was proposed by an XSB expert after debugging XSB's slow performance on the original rules and led to drastic speedups. This change to the rules has a similar impact on SWI-Prolog's running time, so we only show its running times on the drastically faster variant. This change does not affect the running times of the other systems.

Additionally, for loading facts, we experimented with four ways in XSB: predicate `consult/1` for reading a Prolog file, `load_dyn/1` for faster loading of dynamic code, and more specialized versions `load_dync/1` and `load_dynca/1` recommended by an XSB expert. We also experimented with two ways of loading facts in SWI-Prolog: predicate `consult/1`, and 22 lines of specialized handwritten code recommended by an SWI-Prolog expert.

Figure 2-left shows that our straightforward implementation in DistAlgo is much slower than XSB on the original rules but much faster than DLV. All other XSB, SWI-Prolog, clingo, and DistAlgo programs are too fast to tell apart from 0 seconds with this range of CPU times, regardless of optimizations of rules and methods of loading facts.

Figure 2-right shows that our incrementalized implementation in DistAlgo is the fastest; followed by XSB and SWI-Prolog on the reversed rules using `load_dync/1` and using specialized code by hand, respectively, for loading facts; then by clingo; and then by XSB and SWI-Prolog on the reversed rules using other ways to load facts; finally, XSB on the original rules is significantly slower. Results for XSB using `load_dynca/1` are not shown, because they are basically the same as the results using `load_dync/1`.

For XSB on the original rules, only the results using `consult/1` are shown; the results using `load_dyn/1` and `load_dync/1` are similar to each other and about 15–25% slower than using `consult/1`—this is opposite to those times being faster for the reversed rules because (1) compared to XSB times on the original rules, all ways of loading facts are too fast, being a part of XSB times on the reversed rules, as shown in Figure 2-left, and (2) unlike `load_dyn`, etc., `consult/1` statically compiles the facts to support more efficient hashing, showing its advantage for more expensive queries.

Beyond these performance results, our experiments show that significant expertise is needed to achieve high performance with XSB and SWI-Prolog.

Double-win game. The result for this problem was partly unexpected, because it revealed that, for many of our benchmarks, both XSB and SWI-Prolog gave incorrect results. Both clingo and DLV are known to not compute 3-valued semantics. Therefore, we do not compare with these systems for performance. The performance of our DistAlgo implementation for this problem is similar to that for the company control problem but is faster because it involves fewer sets and operations.

For running the double-win game problem in XSB, we were helped by an XSB expert. The result is that when querying using `dwin(X)` in XSB to compute true, false, and undefined values for all positions, the XSB implementation produced different results than our DistAlgo implementation on many of our benchmarks. Some results differ very slightly but some others differ significantly.

However, many logic rule engines also allow the same rules to be directly used for different queries. For all benchmarks that XSB produced different results than ours, when querying using `dwin(x)` for each of the positions `x` separately, XSB produced the same true, false, or undefined values for all positions as our results.

So this experiment helps confirm the correctness of our results and at the same time revealed a bug in XSB. Some careful investigation into this bug indicated that it is a nasty one. After learning that SWI-Prolog added similar support as XSB for computing WFS, we also ran tests in SWI-Prolog and found various incorrect results. Both XSB and SWI-Prolog have since found and fixed bugs that caused these incorrect results.

We also ran the double-win example in both clingo and DLV. Both exited with an error message

that the rule is unsafe.

Note that 2-valued semantics such as extensions of SMS do not give the desired semantics for this game. For example, even on the simple input with only two `move` facts, `move(1,1)` and `move(1,2)`, any such semantics would give `{}`, i.e., no model, meaning that the game is inconsistent, but the desired result is that `dwin(2)` is false, and `dwin(1)` is undefined, i.e., `2` is a losing position, and `1` is a draw position.

Over-win game. The result for this problem was totally unexpected, because we found that not only both XSB and SWI-Prolog gave incorrect query results, but also their developers determined that the inference they do to compute their intended semantics fundamentally cannot compute the correct results. Both clingo and DLV again cannot compute the desired 3-valued semantics, as for the double-win game.

For XSB and SWI-Prolog, their inference to compute WFS deals with loops through explicit negation, but the recursive appearance of `owin` is not explicitly negated. Additionally, their inference does not correctly handle different comparisons that use the results of the same aggregation. These issues are being studied by the XSB team, and an entirely new inference algorithm, based on our founded semantics for computing WFS, is being developed. Our DistAlgo programs for experiments have been used to generate test cases, especially on larger data, because no other known implementations can compute correct results.

9 Related work and conclusion

The study of recursive rules with negation goes back at least to Russell's paradox, discovered over 120 years ago [ID20]. Many logic languages and disagreeing semantics have since been proposed, with significant complications and challenges described in various survey and overview articles, e.g., [AB94, RU95, Fit02, Tru18], and in works on relating and unifying different semantics, e.g., [Dun92, Prz94, Sch95, LZ04, DT08, HDCD10, BDT16, LS20a].

Recursive rules with aggregation have been a subject of study soon after rules with negation were used in programming. They received an even larger variety of disagreeing semantics in 20 years, e.g., [KS91, VG92, SSRB93, CM93, RS97, SNS02, Gel02, MT04, MR04, PDB07, SPT07, FPL$^+$08, LPST10, FPL11, Fer11], and even more intensive studies in the last few years, e.g., [GZ14, SYZ15, AFG15, AL15, Alv16, AFG16, ZR16, GZ17, ZYD$^+$17, ADM18, CFDCP18, GZ18, CFSS19, GZ19, GWM$^+$19, DLW$^+$19, ZDG$^+$19, WZW$^+$20, VBD21], especially as they are needed in graph analysis and machine learning applications.

Aggregation is even more challenging than negation, when used in recursion, because it is more general. For example, the count of all values `x` for which `p(x)` holds is 0 iff for all `x`, `p(x)` does not hold, but the count can be, say, 5, meaning that `p(x)` holds for some 5 values, but does not for the other values, with many possibilities. Even more different and more sophisticated semantics have been proposed for aggregation than for negation.

Major related works are as shown in Table 1, right column. They give disagreeing semantics with each other, without simple formal explanations for the disagreement, as explained there. More detailed comparisons with work by Kemp and Stuckey [KS91], Van Gelder [VG92], Pelov, Denecker, and Bruynooghe [PDB07], Faber, Pfeifer, and Leone [FPL11], Gelfond and Zhang [GZ19], and Hella et al. [HLNW99, HLNW01] appear in Section 6. Among all, Pelov et al.'s work [PDB07], recently reworked for Answer Set Programming (ASP) [VBD22], which uses SMS, is notable for proposing a framework that can be instantiated to extend several prior semantics to handle aggregation. They develop several separate extended semantics. In contrast, our approach uses simple predicate declarations to capture different assumptions made by different semantics in a unifying single semantics.

Many other different semantics have been studied, all focused on restricted classes or issues. The survey

by Ramakrishnan and Ullman [RU95] discusses some different semantics, optimization methods, and uses of recursive rules with aggregation in earlier projects. Ross and Sagiv [RS97] studies monotonic aggregation but not general aggregation. Beeri et al. [BRSS92] presents the valid model semantics for logic programs with negation, set expressions, and grouping, but not aggregation. Sudarshan et al. [SSRB93] extends the valid model semantics for aggregation, gives semantics for more programs than Van Gelder [VG92], and subsumes a class of programs in Ganguly et al. [GGZ91], but it is only a 3-valued semantics. Hella et al. [HLNW99, HLNW01] study expressiveness of aggregation operators but without recursion. Liu et al. [LPST10] give a semantics for logic programs with abstract constraints, which can represent aggregates, and show that, for positive programs, it agrees with one of Pelov et al.'s semantics [PDB07]. A number of other works have followed Gelfond and Zhang's line of study for ASP [CFDCP18, CFSS19, GZ19].

Zaniolo et al. [GGZ91, ZAO93, ZYD+17, GWM+19, DLW+19, ZDG+19] study recursive rules with aggregation for database applications, especially including for big data analysis and machine learning applications in recent years. They study optimizations that exploit monotonicity as well as additional properties of the aggregation operators in computing the least fixed point, yielding superior performance and scalability necessary for these large applications. They discuss insight from their application experience as well as prior research centering on fixed-point computation [ZYD+17], which essentially corresponds to the assumption that predicates are certain.

Our founded semantics and constraint semantics for recursive rules with aggregation unify different previous semantics by allowing different underlying assumptions to be easily specified explicitly, and furthermore separately for each predicate if desired. Our semantics are also fully declarative, giving both a single 3-valued model from simply least fixed-point computation and a set of 2-valued models from simply constraint solving.

The key enabling ideas of simple binary choices for expressing assumptions and simple lease fixed-point computation and constraint solving are taken from Liu and Stoller [LS18, LS20a], where they present a simple unified semantics for recursive rules with negation and quantification. To use the power of founded semantics and constraint semantics in programming, they propose DA logic [LS20b, LS21], for design and analysis logic, that allows different assumptions to be specified as one of four meta-constraints, allows the resulting semantics to be referenced directly, and allows programs to be easily and modularly specified by using knowledge units.

Our semantics can be extended for rules with negation in the conclusion, in the same way as in [LS18, LS20a]. It can also easily be extended for hypotheses that are equalities or negated equalities between variables and constants, because such hypotheses are already used in presenting the semantics. Additionally, DA logic [LS20b, LS21] can be extended to include aggregation, because aggregations and comparisons are interpreted orthogonally from the additional features in DA logic.

There are many directions for future research, including additional language features, efficient implementation methods, and precise complexity guarantees [LS09, TL10, TL11] when possible.

Acknowledgement. We would like to thank David S. Warren and Jan Wielemaker for their excellent help with using XSB and SWI-Prolog.

References

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

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

[ACD+17] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, and J. Zangari. The ASP system DLV2. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 215–221. Springer, 2017.

[ADM18] M. Alviano, C. Dodaro, and M. Maratea. Shared aggregate sets in answer set programming. *Theory and Practice of Logic Programming*, 18(3-4):301–318, 2018.

[AFG15] M. Alviano, W. Faber, and M. Gebser. Rewriting recursive aggregates in answer set programming: back to monotonicity. *Theory and Practice of Logic Programming*, 15(4-5):559–573, 2015.

[AFG16] M. Alviano, W. Faber, and M. Gebser. From non-convex aggregates to monotone aggregates in ASP. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 4100–4104, 2016.

[AL15] M. Alviano and N. Leone. Complexity and compilation of GZ-aggregates in answer set programming. *Theory and Practice of Logic Programming*, 15(4-5):574–587, 2015.

[Alv16] M. Alviano. Evaluating answer set programming with non-convex recursive aggregates. *Fundamenta Informaticae*, 149(1-2):1–34, 2016.

[BDT16] M. Bruynooghe, M. Denecker, and M. Truszczynski. First order logic with inductive definitions for model-based problem solving. *AI Magazine*, 37(3):69–80, 2016.

[BRSS92] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 91–104, 1992.

[CFDCP18] P. Cabalar, J. Fandinno, L. F. Del Cerro, and D. Pearce. Functional ASP with intensional sets: Application to Gelfond-Zhang aggregates. *Theory and Practice of Logic Programming*, 18(3-4):390–405, 2018.

[CFSS19] P. Cabalar, J. Fandinno, T. Schaub, and S. Schellhorn. Gelfond-zhang aggregates as propositional formulas. *Artificial Intelligence*, 274:26–43, 2019.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

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

[CM93] M. P. Consens and A. O. Mendelzon. Low-complexity aggregation in GraphLog and Datalog. *Theoretical Computer Science*, 116(1):95–116, 1993.

[DLW+19] A. Das, Y. Li, J. Wang, M. Li, and C. Zaniolo. Bigdata applications from graph analytics to machine learning by aggregates in recursion. In *Proceedings of the 35th International Conference on Logic Programming (Technical Communications)*, pages 273–279, 2019.

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

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

[Fer11] P. Ferraris. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic*, 12(4):1–40, July 2011.

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

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

[FPL+08] W. Faber, G. Pfeifer, N. Leone, T. Dell'Armi, and G. Ielpa. Design and implementation of aggregate functions in the DLV system. *Theory and Practice of Logic Programming*, 8(5-6):545–580, 2008.

[FPL11] W. Faber, G. Pfeifer, and N. Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.

[Gel02] M. Gelfond. Representing knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*, pages 413–451. Springer, 2002.

[GGZ91] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–163, 1991.

[GL88] M. Gelfond and V. 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.

[GWM+19] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. RaSQL: Greater power and performance for big data analytics with recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data*, pages 467–484, 2019.

[GZ14] M. Gelfond and Y. Zhang. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming*, 14(4-5):587–601, 2014.

[GZ17] M. Gelfond and Y. Zhang. Vicious circle principle and formation of sets in ASP based languages. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 146–159. Springer, 2017.

[GZ18] M. Gelfond and Y. Zhang. Vicious circle principle and logic programs with aggregates. *Computing Research Repository*, cs.AI(arXiv:1808.07050), 2018.

[GZ19] M. Gelfond and Y. Zhang. Vicious circle principle, aggregates, and formation of sets in ASP based languages. *Artificial Intelligence*, 275:28–77, Oct. 2019.

[HDCD10] P. Hou, B. De Cat, and M. Denecker. FO(FD): Extending classical logic with rule-based fixpoint definitions. *Theory and Practice of Logic Programming*, 10(4-6):581–596, 2010.

[HLNW99] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 35. IEEE Computer Society, 1999.

[HLNW01] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *Journal of the ACM*, 48(4):880–907, 2001.

[ID20] A. D. Irvine and H. Deutsch. Russell's paradox. *Stanford Encyclopedia of Philosophy*, 2020. `https://plato.stanford.edu/entries/russell-paradox/` First published Fri Dec 8, 1995; substantive revision Mon Oct 12, 2020. Accessed Jan 3, 2021.

[KS91] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Symposium on Logic Programming*, pages 387–401, 1991.

[LBSL16] Y. A. Liu, J. Brandvein, S. D. Stoller, and B. Lin. Demand-driven incremental object queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 228–241. ACM Press, 2016.

[Liu18] Y. A. Liu. Logic programming applications: What are the abstractions and implementations? In M. Kifer and Y. A. Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, chapter 10, pages 519–557. ACM and Morgan & Claypool, 2018.

[LPST10] L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174(3):295–315, 2010.

[LS09] Y. A. Liu and S. 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] Y. A. Liu and S. D. Stoller. Founded semantics and constraint semantics of logic rules. In *Proceedings of the 2018 International Symposium on Logical Foundations of Computer Science*, volume 10703 of *Lecture Notes in Computer Science*, pages 221–241. Springer, Jan. 2018.

[LS20a] Y. A. Liu and S. D. Stoller. Founded semantics and constraint semantics of logic rules. *Journal of Logic and Computation*, 30(8):1609–1638, Dec. 2020. Also `http://arxiv.org/abs/1606.06269`.

[LS20b] Y. A. Liu and S. D. Stoller. Knowledge of uncertain worlds: Programming with logical constraints. In *Proceedings of the 2020 International Symposium on Logical Foundations of Computer Science*, volume 11972 of *Lecture Notes in Computer Science*, pages 111–127. Springer, Jan. 2020.

[LS21] Y. A. Liu and S. D. Stoller. Knowledge of uncertain worlds: Programming with logical constraints. *Journal of Logic and Computation*, 31(1):193–212, Jan. 2021. Also `https://arxiv.org/abs/1910.10346`.

[LS22] Y. A. Liu and S. D. Stoller. Recursive rules with aggregation: A simple unified semantics. In *Proceedings of the 2022 International Symposium on Logical Foundations of Computer Science*, volume 13137 of *Lecture Notes in Computer Science*, pages 156–179. Springer, Jan. 2022. Also `http://arxiv.org/abs/2007.13053`.

[LSL17] Y. A. Liu, S. D. Stoller, and B. Lin. From clarity to efficiency for distributed algorithms. *ACM Transactions on Programming Languages and Systems*, 39(3):12:1–12:41, May 2017.

[LZ04] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 264–277. Morgan Kaufmann, 1990.

[MR04] V. W. Marek and J. B. Remmel. Set constraints in logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 167–179. Springer, 2004.

[MT04] V. W. Marek and M. Truszczynski. Logic programs with abstract constraint atoms. In *Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence*, pages 86–91. AAAI Press / The MIT Press, 2004.

[PDB07] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7(3):301–353, 2007.

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

[RS92] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 114–126, 1992.

[RS97] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

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

[Sch95] J. S. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[SPT07] T. C. Son, E. Pontelli, and P. H. Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. *Journal of Artificial Intelligence Reseearch*, 29:353–389, 2007.

[SSRB93] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri. Extending the well-founded and valid semantics for aggregation. In *Proceedings of the 1993 International Symposium on Logic programming*, pages 590–608, 1993.

[SWS+17] T. Swift, D. S. Warren, K. Sagonas, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, S. Dawson, and M. Kifer. *The XSB System Version 3.8.x*, Oct. 2017. `http://xsb.sourceforge.net`.

[SYZ15] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering*, pages 867–878, 2015.

[TL10] K. T. Tekle and Y. A. Liu. Precise complexity analysis for efficient Datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 35–44, 2010.

[TL11] K. T. Tekle and Y. A. Liu. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 661–672, 2011.

[Tru18] M. Truszczynski. An introduction to the stable and well-founded semantics of logic programs. In M. Kifer and Y. A. Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 121–177. ACM and Morgan & Claypool, 2018.

[VBD21] L. Vanbesien, M. Bruynooghe, and M. Denecker. Analyzing semantics of aggregate answer set programming using approximation fixpoint theory. *Computing Research Repository*, cs.AI(arXiv:2104.14789), 2021.

[VBD22] L. Vanbesien, M. Bruynooghe, and M. Denecker. Analyzing semantics of aggregate answer set programming using approximation fixpoint theory. *Theory and Practice of Logic Programming*, 22(4):523–537, 2022.

[VG86] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *Proceedings of the 3rd IEEE-CS Symposium on Logic Programming*, pages 127–138, 1986.

[VG92] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 127–138, June 2-4, 1992, San Diego, California, 1992.

[VG93] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VGRS88] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

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

[WSTL12] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[WZW+20] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of data*, pages 2439–2454, 2020.

[YSZ17] M. Yang, A. Shkapsky, and C. Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *The VLDB Journal*, 26(2):229âĂŞ248, Apr. 2017.

[ZAO93] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: The LDL++ approach. In *International Conference on Deductive and Object-Oriented Databases*, pages 204–221. Springer, 1993.

[ZDG+19] C. Zaniolo, A. Das, J. Gu, Y. Li, M. Li, and J. Wang. Monotonic properties of completed aggregates in recursive queries. *Computing Research Repository*, cs.DB(arXiv:1910.08888), 2019.

[ZR16] Y. Zhang and M. Rayatidamavandi. A characterization of the semantics of logic programs with aggregates. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1338–1344, 2016.

[ZYD+17] C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory and Practice of Logic Programming*, 17(5-6):1048–1065, 2017.

A Additional proofs

Proof of Theorem 1. The proof of consistency of the founded model is an extension of the corresponding proof by induction for the language without aggregation [LS20a, Theorem 1]; for brevity, we refer to that proof as the original proof. The proof is by induction on the sequence of interpretations constructed

in the semantics by steps that either apply one-step derivability or add negated self-false atoms. Two extensions to the proof are needed to show consistency for the language extended with aggregation.

To show that steps that apply one-step derivability still preserve consistency, we extend the original proof to show consistency for comparison literals, in the sense that $DC(\pi, I)$ cannot contain a comparison literal and its complement; this result is used, together with the result in the original proof that I cannot contain a predicate literal and its complement, to show that the body of a rule and its negation cannot both be true in I. This result follows directly from the definition of linear-time derivability in Figure 1: for each pair of biconditionals for deriving complementary comparisons, the right sides of those biconditionals are mutually exclusive conditions, i.e., the conjunction of those two conditions is not satisfiable.

To show that steps that add negated self-false atoms still preserve consistency, we extend the proof to show that the extended definition of unfounded set still ensures that none of the atoms in an unfounded set U for an interpretation I are derivable in $I \cup \neg \cdot U$. This property still holds because the definition ensures that, for each rule instance R that could be used to derive an atom in U, (1) some hypothesis of R is false in I and hence is false in $I \cup \neg \cdot U$, (2) some positive predicate hypothesis of R is in U and hence is false in $I \cup \neg \cdot U$, or (3) some comparison hypothesis of R is false in $I \cup \neg \cdot U$. Note that these three cases correspond to the three cases in the extended definition of unfounded set.

For consistency of constraint semantics, note that constraint models are required to be interpretations, which are consistent by definition. ■

Proof of Theorem 2. The proof that *Founded*(π) is a model of π and *Cmpl*(π) is an extension of the corresponding proof for the language without aggregation [LS20a, Theorem 2]; for brevity, we refer to that proof as the original proof. The original proof relies on the result that the body of a ground instance $A \leftarrow B$ of a rule in *Cmpl*(π) cannot become true in *AddNeg* for an SCC C, i.e., as a result of adding negative literals for certain predicates in C to the interpretation, where C is the SCC containing the predicate Q in the conclusion A. This result is shown in the original proof by proving by contradiction that B cannot contain a negative literal for a certain predicate in C. To show that this result holds in the language with aggregation, we prove by contradiction that adding those negative literals to the interpretation cannot cause a comparison atom in B to become derivable.

We suppose that B contains a comparison atom A that becomes derivable and therefore true in *AddNeg* for C and show a contradiction. Since A becomes true in *AddNeg* for C, A must contain a non-positive occurrence of a certain predicate P in C. P and Q are in the same SCC C, 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 C, and Q depends non-positively on P, Q has a circular non-positive dependency, so Q must be uncertain, a contradiction.

Constraint models are 2-valued models of *Cmpl*(π) by definition. Any model of *Cmpl*(π) is also a model of π, because π is logically equivalent to the subset of *Cmpl*(π) obtained by removing the completion rules added by *AddInv*. ■

Proof of Theorem 4. First, we show that *Founded*(π) is 2-valued for predicates in C. Let RS be the set of all ground instances of combined rules and completion rules in *Cmpl*(π) for predicates in C. Note that every positive literal and negative literal for every predicate in C appears as the conclusion of at least one rule in G (this holds even if rules in π contain conclusions of forms such as `p(x,x)` or `p(x,0)`, due to the fresh variables and existential quantifiers introduced by *Combine*). Let UA be the set of ground predicate atoms for predicates in C that are undefined in *Founded*$_0$(π). For each atom A in UA, since the predicate in A is complete and A is not true or false in *Founded*$_0$(π), (1) for every rule R in RS with conclusion A, some hypothesis of R is false or undefined in *Founded*$_0$(π), and (2) for some rule R in RS with conclusion A, some hypothesis of R is undefined in *Founded*$_0$(π). Since all predicates

in SCCs that precede C are certain, these undefined hypotheses must be atoms in UA or their negations, or comparisons whose outcomes depend on the truth value of 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, $\neg B$, or a comparison whose outcome depends on the truth value of 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 C do not have circular non-positive dependency, this implies that all dependencies in the cycles are positive. These observations, together with all predicates in C being closed, imply that the predicate literals in every cycle, and hence all atoms in UA, are in $\mathit{SelfFalse}_{\pi}(\mathit{Founded}_0(\pi))$. Note any comparisons that give rise to the dependencies in the cycles becomes false when all atoms in UA are false (and hence satisfy clause (3) in the definition of unfounded set), because the comparison must be true or false when all literals it depends on are true or false, and setting literals on which it has positive dependence to false cannot make the comparison become true. This implies that $\mathit{Founded}(\pi)$ contains the negations of all literals in UA. Therefore, $\mathit{Founded}(\pi)$ is 2-valued.

Next, we show that $\mathit{Founded}(\pi) = \mathit{Founded}(\pi')$. For each predicate P in C, 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 $\mathit{Founded}(\pi)$ and $\mathit{Founded}(\pi')$ 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 $\mathit{Constraint}(\pi) = \mathit{Constraint}(\pi')$. We consider the conditions in the definition of *Constraint*, in turn.

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

Consider the second condition, namely, M is a model of $\mathit{Cmpl}(\pi)$. It differs for the two programs in that $\mathit{Cmpl}(\pi)$ contains combined rules and completion rules for predicates in C, while $\mathit{Cmpl}(\pi')$ contains the original rules in π for predicates in C. Since $\mathit{Founded}(\pi) = \mathit{Founded}(\pi')$, Theorem 2 implies $\mathit{Founded}(\pi)$ is a model of both $\mathit{Cmpl}(\pi)$ and $\mathit{Cmpl}(\pi')$. Since $\mathit{Founded}(\pi)$ is 2-valued for predicates in C and all predicates on which they depend, it is 2-valued for all predicates used in those rules. Therefore, every interpretation M that is a superset of $\mathit{Founded}(\pi)$ contains the same predicate literals as $\mathit{Founded}(\pi)$ for all predicates used in those rules and hence is also a model of both $\mathit{Cmpl}(\pi)$ and $\mathit{Cmpl}(\pi')$. Since M is (by the first condition) a superset of $\mathit{Founded}(\pi)$, and it is a model of both $\mathit{Cmpl}(\pi)$ and $\mathit{Cmpl}(\pi')$. Thus, the second condition is equivalent for the two programs.

Consider the third condition, namely, there does not exist a non-empty subset S of $M \setminus \mathit{Founded}(\pi)$ containing only positive literals for closed predicates and such that $S = \mathit{SelfFalse}_{\pi}(M \setminus S)$. Since $\mathit{Founded}(\pi)$ is 2-valued for predicates in C, and $S \subseteq M \setminus \mathit{Founded}(\pi)$, S cannot contain literals for predicates in C; furthermore, this and the disjointness condition in the definition of unfounded set imply that $\mathit{SelfFalse}_{\pi}(M \setminus S)$ cannot contain literals for predicates in C. The same conclusions hold for π', since $\mathit{Founded}(\pi) = \mathit{Founded}(\pi')$. Since S and $\mathit{SelfFalse}_{\pi}(M \setminus S)$ do not contain literals for predicates in C, and π and π' have the same rules and declarations for predicates not in C, $\mathit{SelfFalse}_{\pi}(M \setminus S) = \mathit{SelfFalse}_{\pi'}(M \setminus S)$. Thus, the third condition is equivalent for the two programs. ■