Recursive Rules with Aggregation: A Simple Unified Semantics*

Yanhong A. Liu and Scott D. Stoller

Computer Science Department, Stony Brook University, Stony Brook, NY, USA
{liu,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 counts and sums 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 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 applied our semantics to a wide range of challenging examples, and performed experiments on the most challenging ones, all confirming our analyzed results.

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 [39]. 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 counts and sums. 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.

* 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.

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 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 [35]. Many different semantics, which disagree with each other, have been studied for recursive rules with negation, as summarized in Section 6. Two of them, well-founded semantics (WFS) [66, 64] and stable model semantics (SMS) [26], 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 6, especially as they are needed in graph analysis and machine learning applications. However, the many different semantics proposed, e.g., [63, 30], are even more intricate than WFS and SMS for recursive rules with negation, and include even different extensions for WFS, e.g., [36, 63, 59], and for SMS, e.g., [36, 47, 30]. Some authors also changed their own minds about the desired semantics, e.g., [25, 30]. 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 [41, 42]. 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 compared with main prior semantics for rules with aggregation, and showed how our semantics is direct and follows precisely from usual meanings of aggregations. We further applied our semantics to a wide range of challenging examples, and showed that our semantics is simple and matches the desired semantics in all cases. Additionally, we performed experiments on two most challenging examples,

confirming the correctness of our computed results, while also discovering worse performance and some wrong results from well-known systems. Additional results from these comparisons, examples, and experiments are described in [44].

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. 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., [19, 27, 29]. 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 [36] and Gelfond [25], 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)` being true.
- **One model: Faber et al. 2011, Gelfond-Zhang 2014-2019.** According to Faber, Pfeifer, and Leone [19] and Gelfond and Zhang [27, Examples 2 and 7], [29, Examples 4 and 6], and [30, Example 9], the rule above has only one model: the empty model.

As one of the several main efforts investigating aggregation, Gelfond and Zhang [25, 27, 73, 28–30] have studied the challenges and solutions extensively, presenting dozens of definitions and propositions and discussing dozens of examples [30]. 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.

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 [41, 42] 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 stated in general in Section 4.5.

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:

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

  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 [41, 42]. 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 [44].

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, 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 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

Declarations	Semantics	Extending	Reference	Prior Extensions
certain	Founded, Constraint	Stratified (Perfect)	Van Gelder 1986 [62]	e.g., [49, 53]
uncertain, not complete	Founded	(none found)		(none found)
	Constraint	First-Order Logic		e.g., [33]
uncertain, complete, not closed	Founded	Fitting (Kripke-Kleene)	Fitting 1985 [22]	Pelov et al. 2007 [50]
	Constraint	Supported	Apt et al. 1988 [7]	Pelov et al. 2007 [50]
uncertain, complete, closed	Founded	WFS	Van Gelder et al. 1988 [65, 66]	Kemp-Stuckey 1991 [36] Van Gelder 1992 [63] Pelov et al. 2007 [50]
	Constraint	SMS	Gelfond-Lifschitz 1988 [26]	Kemp-Stuckey 1991 [36] Pelov et al. 2007 [50] Faber et al. 2011 [19] Gelfond-Zhang 2014-2019 [30]

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.

computations are exact. Our semantics detects and can report cases where an inference is blocked because it involves a value outside the range $[-NRB, NRB]$; for details, see the description of range-blocked inference in [44].

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 $\neg$:

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

Q and P_i's are predicates, and each argument X_k and X_{ij} is a constant or a variable. In arguments of predicates in examples, we use numbers and quoted strings for constants and letters for variables.

If $h = 0$, there are no P_i's or X_{ij}'s, and each X_k must be a constant, in which case $Q(X_1, ..., X_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case the left side of $\leftarrow$ is called the *conclusion*, the right side is called the *body*, and each conjunct in the body is called a *hypothesis*. Note that 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 as well as conjunction. Conjunction and disjunction may be nested arbitrarily.

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

$$\exists\ X_1, ..., X_a \mid B \quad \text{existential quantification}$$
$$\forall\ X_1, ..., X_a \mid B \quad \text{universal quantification}$$

where each X_i is a variablethat appears 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.

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., `max {Y: ∃ X | p(X,Y)} > 0` is rewritten to `max {Y: q(Y)} > 0` together with `q(Y)` ← `∃ X | p(X,Y)`. Each auxiliary predicate has default declarations, except that it is declared closed if some predicate in the body of the rule defining the auxiliary predicate is declared closed.

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).

Note that negation applied to comparisons can be eliminated by reversing the comparison operators; for example, the negation of a comparison using $\leq$ is a comparison using $>$.

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. Our principled approach can easily support additional aggregation and comparison functions, e.g., on other data types such as strings.

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 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 containing a predicate atom or comparison atom is called a *predicate literal* or *comparison literal*, respectively. 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.

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; otherwise, the occurrence is a *non-positive occurrence*.

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 a syntax for predicate declarations, because it is straightforward, and most examples use default declarations. However, the language in [43, 45] supports such declarations.

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 [41, 42] 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 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

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

Fig. 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 $\leq k$.

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 $\mathit{agg}\, S \odot k$ is *derivable* in interpretation I of π, denoted $\pi, I \vdash \mathit{agg}\, S \odot k$, 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 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 [22] (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 $\mathit{Founded}_0$, 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 $\mathit{Founded}_0(\pi)$, 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 $Cmpl(\pi)$ returns the *completed program* of π. Formally, $Cmpl(\pi) = AddInv(Combine(\pi))$, where *Combine* and *AddInv* are defined as follows.

The function $Combine(\pi)$ returns the program obtained from π by replacing the facts and rules defining each uncertain complete predicate Q with a single *combined rule* for Q, defined as follows. 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 $AddInv(\pi)$ returns the program obtained from π by adding, for each uncertain complete predicate Q, a *completion rule* that derives negative literals for Q. The completion rule for Q is obtained from the inverse of the combined rule defining Q (recall that the inverse of $A \leftarrow B$ 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) eliminate 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 [14] and Fitting semantics [22].

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 $NameNeg(\pi)$ 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 in $\neg P(X_1, ..., X_a)$ 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 $P(X_1, ..., X_a)$ (where negative occurrence is defined symmetrically as positive occurrence) to allow negative conclusions to be derived and used as facts. We replace any negative occurrence of $P(X_1, ..., X_a)$ not in $\neg P(X_1, ..., X_a)$ with $\neg\,\texttt{n}.P(X_1, ..., X_a)$ also to use these facts. Other occurrences, 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

$\mathtt{n}.P(X_1, ..., X_a)$ and $P(X_1, ..., X_a)$ follow the one-step derivability. We have not seen any example that needs this, 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 earlier SCCs do not depend on later ones; it is easy to show that any linearization of the dependency order leads to the same result for *LFPbySCC*. 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 *LFPbySCC*(π) $= I_n$, where $I_0 = \emptyset$ and $I_i =$ *AddNeg*(*LFP*($T_{Proj(\pi,C_i)\cup I_{i-1}}$), C_i) for $i \in 1..n$. *LFP* is the least fixed point operator. *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 $\mathtt{n}.P(v_1, ..., v_a)$. The least fixed point is well-defined, because the one-step derivability function $T_{Proj(\pi,C_i)\cup I_{i-1}}$ is monotonic with respect to $\subseteq$, i.e., for all interpretations J and J', $T_{Proj(\pi,C_i)\cup I_{i-1}}(J) \subseteq T_{Proj(\pi,C_i)\cup I_{i-1}}(J')$ whenever $J \subseteq J'$; the proof is straightforward [44].

The function *UnNameNeg*(I) returns the interpretation obtained from interpretation I by replacing each atom $\mathtt{n}.P(X_1, ..., X_a)$ 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 [66] 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 value 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 in I. 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 [66, 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 $S = \mathit{SelfFalse}_\pi(M \setminus S)$. Intuitively, 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 can be set to false.

We also require that an interpretation that leads to an error in a comparison is 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 be made less strict about errors.

Note that condition (3) differs from the corresponding condition in constraint semantics without aggregation [41, 42], 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 $\mathit{Founded}(\pi)$, satisfying condition (1), and then checking whether the other conditions in the definition of constraint model are satisfied.

4.5 Properties of the semantics

We briefly state several important properties of the semantics; detailed statements and proofs are in [44]. (1) *Consistency:* The founded model and constraint models

of a program π are consistent. (2) *Correctness:* 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 $Cmpl(\pi)$. (3) *Same SCC, same certainty:* All predicates in an SCC have the same certainty. (4) *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. (5) *Equivalent declarations:* Changing predicate declarations from uncertain, complete, and closed to certain when allowed, or vice versa, preserves founded and constraint semantics.

5 Examples: company control and double win

We discuss the well-known challenging company control problem [13, 54, 19, 30] and an even more challenging game problem that generalizes the well-known win-not-win game [41, 42].

5.1 Company control—a well-known challenge

This is Examples 1.1 and 2.13 in [19] and is also used in Example 12 in [30]. The problem was also discussed repeatedly before [49, 36, 63, 54, 50] and earlier [13]. 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 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 [19].

5.2 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 [41, 42], 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.

5.3 Experiments

We also performed experiments with our new semantics. We implemented straightforward and incremental least fixed-point computations for example problems in DistAlgo [46], an extension of Python. We also compared with results computed by three systems that support negation and aggregation in recursion: XSB [60], the most well-known such system that computes WFS, and clingo [4] and DLV [20, 2], the most well-known such systems that compute SMS.

For the company control problem, our incremental program in DistAlgo (v.1.1.0b15 on Python 3.7) was the fastest; followed by clingo (v.5.4.0), about 7 times slower; followed by XSB (v.3.8.0), our straightforward program in DistAlgo, and DLV (`https://www.dbai.tuwien.ac.at/proj/dlv/dlvRecAggr/` (accessed 2020-09-21))[1], each

[1] 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 `http://www.dlvsystem.com/dlv/` (last accessed 2021-11-04).

asymptotically and drastically slower than the preceding one. Most recent investigation found that changing the order of hypotheses in rules in XSB can improve the running times for this problem asymptotically.

For the double win problem, clingo and DLV cannot compute the desired 3-valued semantics, and XSB was found to compute incorrect results on some of our benchmarks. Most recent investigation found that SWI-Prolog [69] added support for computing WFS, but was found to compute incorrect results for this problem on some smallest inputs. Both SWI-Prolog and XSB have since found and fixed bugs that caused these incorrect results.

6 Related work and conclusion

The study of recursive rules with negation goes back at least to Russell's paradox, discovered over 120 years ago [35]. Many logic languages and disagreeing semantics have since been proposed, with significant complications and challenges described in various survey and overview articles, e.g., [8, 52, 23, 61], and in works on relating and unifying different semantics, e.g., [18, 51, 55, 37, 17, 34, 10, 42].

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., [36, 63, 59, 15, 54, 57, 25, 48, 47, 50, 58, 20, 38, 19, 21], and even more intensive studies in the last few years, e.g., [27, 56, 4, 6, 1, 5, 73, 28, 72, 3, 11, 29, 12, 30, 31, 16, 71, 68, 67], especially as they are needed in graph analysis and machine learning applications.

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 [36], Van Gelder [63], Pelov, Denecker, and Bruynooghe [50], Faber, Pfeifer, and Leone [19], Gelfond and Zhang [30], and Hella et al. [32, 33] appear in [44]. Among all, Pelov et al.'s work [50], recently reworked for ASP [67], 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 [52] discusses some different semantics, optimization methods, and uses of recursive rules with aggregation in earlier projects. Ross and Sagiv [54] studies monotonic aggregation but not general aggregation. Beeri et al. [9] presents the valid model semantics for logic programs with negation, set expressions, and grouping, but not aggregation. Sudarshan et al. [59] extends the valid model semantics for aggregation, gives semantics for more programs than Van Gelder [63], and subsumes a class of programs in Ganguly et al. [24], but it is only a 3-valued semantics. Hella et al. [32, 33] study expressiveness of aggregation operators but without recursion. Liu et al. [38] 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 [50]. A number of other works have followed Gelfond and Zhang's line of study for ASP [11, 12, 30].

Zaniolo et al. [24, 70, 72, 31, 16, 71] 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 for centering on fixed-point computation [72], 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 a 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 [41, 42], where they present a simple unified semantics for recursive rules with negation and quantification.

Our semantics can be extended for rules with negation in the conclusion, in the same way as in [41]. 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.

There are many directions for future research, including additional language features, efficient implementation methods, and precise complexity guarantees [40] 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

1. Alviano, M.: Evaluating answer set programming with non-convex recursive aggregates. Fundamenta Informaticae **149**(1-2), 1–34 (2016)
2. Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., Zangari, J.: The ASP system DLV2. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 215–221. Springer (2017)
3. Alviano, M., Dodaro, C., Maratea, M.: Shared aggregate sets in answer set programming. Theory and Practice of Logic Programming **18**(3-4), 301–318 (2018)
4. Alviano, M., Faber, W., Gebser, M.: Rewriting recursive aggregates in answer set programming: back to monotonicity. Theory and Practice of Logic Programming **15**(4-5), 559–573 (2015)
5. Alviano, M., Faber, W., Gebser, M.: From non-convex aggregates to monotone aggregates in ASP. In: Proceedings of the International Joint Conference on Artificial Intelligence. pp. 4100–4104 (2016)
6. Alviano, M., Leone, N.: Complexity and compilation of GZ-aggregates in answer set programming. Theory and Practice of Logic Programming **15**(4-5), 574–587 (2015)

7. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann (1988)
8. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. Journal of Logic Programming **19**, 9–71 (1994)
9. Beeri, C., Ramakrishnan, R., Srivastava, D., Sudarshan, S.: The valid model semantics for logic programs. In: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 91–104 (1992)
10. Bruynooghe, M., Denecker, M., Truszczynski, M.: First order logic with inductive definitions for model-based problem solving. AI Magazine **37**(3), 69–80 (2016)
11. Cabalar, P., Fandinno, J., Del Cerro, L.F., Pearce, D.: Functional ASP with intensional sets: Application to Gelfond-Zhang aggregates. Theory and Practice of Logic Programming **18**(3-4), 390–405 (2018)
12. Cabalar, P., Fandinno, J., Schaub, T., Schellhorn, S.: Gelfond-zhang aggregates as propositional formulas. Artificial Intelligence **274**, 26–43 (2019)
13. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (1990)
14. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum Press (1978)
15. Consens, M.P., Mendelzon, A.O.: Low-complexity aggregation in GraphLog and Datalog. Theoretical Computer Science **116**(1), 95–116 (1993)
16. Das, A., Li, Y., Wang, J., Li, M., Zaniolo, C.: Bigdata applications from graph analytics to machine learning by aggregates in recursion. In: Proceedings of the 35th International Conference on Logic Programming (Technical Communications). pp. 273–279 (2019)
17. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Transactions on Computational Logic **9**(2), 14 (2008)
18. Dung, P.M.: On the relations between stable and well-founded semantics of logic programs. Theoretical Computer Science **105**(1), 7–25 (1992)
19. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence **175**(1), 278–298 (2011)
20. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming **8**(5-6), 545–580 (2008)
21. Ferraris, P.: Logic programs with propositional connectives and aggregates. ACM Transactions on Computational Logic **12**(4), 1–40 (July 2011)
22. Fitting, M.: A Kripke-Kleene semantics for logic programs. Journal of Logic Programming **2**(4), 295–312 (1985)
23. Fitting, M.: Fixpoint semantics for logic programming: A survey. Theoretical Computer Science **278**(1), 25–51 (2002)
24. Ganguly, S., Greco, S., Zaniolo, C.: Minimum and maximum predicates in logic programming. In: Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 154–163 (1991)
25. Gelfond, M.: Representing knowledge in A-Prolog. In: Computational Logic: Logic Programming and Beyond, pp. 413–451. Springer (2002)
26. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference and Symposium on Logic Programming. pp. 1070–1080. MIT Press (1988)
27. Gelfond, M., Zhang, Y.: Vicious circle principle and logic programs with aggregates. Theory and Practice of Logic Programming **14**(4-5), 587–601 (2014)
28. Gelfond, M., Zhang, Y.: Vicious circle principle and formation of sets in ASP based languages. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 146–159. Springer (2017)
29. Gelfond, M., Zhang, Y.: Vicious circle principle and logic programs with aggregates. Computing Research Repository **cs.AI**(arXiv:1808.07050) (2018), `https://arxiv.org/abs/1808.07050`
30. Gelfond, M., Zhang, Y.: Vicious circle principle, aggregates, and formation of sets in ASP based languages. Artificial Intelligence **275**, 28–77 (Oct 2019)
31. Gu, J., Watanabe, Y.H., Mazza, W.A., Shkapsky, A., Yang, M., Ding, L., Zaniolo, C.: 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. pp. 467–484 (2019)

32. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. In: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science. p. 35. IEEE Computer Society (1999)
33. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. Journal of the ACM **48**(4), 880–907 (2001)
34. Hou, P., De Cat, B., Denecker, M.: FO(FD): Extending classical logic with rule-based fixpoint definitions. Theory and Practice of Logic Programming **10**(4-6), 581–596 (2010)
35. Irvine, A.D., Deutsch, H.: 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
36. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: Proceedings of the International Symposium on Logic Programming. pp. 387–401 (1991)
37. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2), 115–137 (2004)
38. Liu, L., Pontelli, E., Son, T.C., Truszczynski, M.: Logic programs with abstract constraint atoms: The role of computations. Artificial Intelligence **174**(3), 295–315 (2010)
39. Liu, Y.A.: Logic programming applications: What are the abstractions and implementations? In: Kifer, M., Liu, Y.A. (eds.) Declarative Logic Programming: Theory, Systems, and Applications, chap. 10, pp. 519–557. ACM and Morgan & Claypool (2018)
40. Liu, Y.A., Stoller, S.D.: From Datalog rules to efficient programs with time and space guarantees. ACM Transactions on Programming Languages and Systems **31**(6), 1–38 (2009)
41. Liu, Y.A., Stoller, S.D.: Founded semantics and constraint semantics of logic rules. In: Proceedings of the 2018 International Symposium on Logical Foundations of Computer Science. Lecture Notes in Computer Science, vol. 10703, pp. 221–241. Springer (Jan 2018)
42. Liu, Y.A., Stoller, S.D.: 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`
43. Liu, Y.A., Stoller, S.D.: Knowledge of uncertain worlds: Programming with logical constraints. In: Proceedings of the 2020 International Symposium on Logical Foundations of Computer Science. Lecture Notes in Computer Science, vol. 11972, pp. 111–127. Springer (Jan 2020)
44. Liu, Y.A., Stoller, S.D.: Recursive rules with aggregation: A simple unified semantics. Computing Research Repository **cs.DB**(arXiv:2007.13053) (July 2020), `http://arxiv.org/abs/2007.13053`
45. Liu, Y.A., Stoller, S.D.: 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`
46. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. ACM Transactions on Programming Languages and Systems **39**(3), 12:1–12:41 (May 2017)
47. Marek, V.W., Remmel, J.B.: Set constraints in logic programming. In: Logic Programming and Nonmonotonic Reasoning. pp. 167–179. Springer (2004)
48. Marek, V.W., Truszczynski, M.: Logic programs with abstract constraint atoms. In: Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence. pp. 86–91. AAAI Press / The MIT Press (2004)
49. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: Proceedings of the 16th International Conference on Very Large Databases. pp. 264–277. Morgan Kaufmann (1990)
50. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory and Practice of Logic Programming **7**(3), 301–353 (2007)
51. Przymusinski, T.C.: Well-founded and stationary models of logic programs. Annals of Mathematics and Artificial Intelligence **12**(3), 141–187 (1994)
52. Ramakrishnan, R., Ullman, J.D.: A survey of deductive database systems. Journal of Logic Programming **23**(2), 125–149 (1995)
53. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. In: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 114–126 (1992)

54. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive databases. Journal of Computer and System Sciences **54**(1), 79–97 (1997)
55. Schlipf, J.S.: The expressive powers of the logic programming semantics. Journal of Computer and System Sciences **51**(1), 64–86 (1995)
56. Shkapsky, A., Yang, M., Zaniolo, C.: Optimizing recursive queries with monotonic aggregates in DeALS. In: Proceedings of the 2015 IEEE 31st International Conference on Data Engineering. pp. 867–878 (2015)
57. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2), 181–234 (2002)
58. Son, T.C., Pontelli, E., Tu, P.H.: Answer sets for logic programs with arbitrary abstract constraint atoms. Journal of Artificial Intelligence Reseearch **29**, 353–389 (2007)
59. Sudarshan, S., Srivastava, D., Ramakrishnan, R., Beeri, C.: Extending the well-founded and valid semantics for aggregation. In: Proceedings of the 1993 International Symposium on Logic programming. pp. 590–608 (1993)
60. Swift, T., Warren, D.S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R.F., Saha, D., Dawson, S., Kifer, M.: The XSB System Version 3.8,x (2017), `http://xsb.sourceforge.net`
61. Truszczynski, M.: An introduction to the stable and well-founded semantics of logic programs. In: Kifer, M., Liu, Y.A. (eds.) Declarative Logic Programming: Theory, Systems, and Applications, pp. 121–177. ACM and Morgan & Claypool (2018)
62. Van Gelder, A.: Negation as failure using tight derivations for general logic programs. In: Proceedings of the 3rd IEEE-CS Symposium on Logic Programming. pp. 127–138 (1986)
63. Van Gelder, A.: The well-founded semantics of aggregation. In: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. pp. 127–138. June 2-4, 1992, San Diego, California (1992)
64. Van Gelder, A.: The alternating fixpoint of logic programs with negation. Journal of Computer and System Sciences **47**(1), 185–221 (1993)
65. Van Gelder, A., Ross, K., Schlipf, J.S.: 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. pp. 221–230 (1988)
66. Van Gelder, A., Ross, K., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3), 620–650 (1991)
67. Vanbesien, L., Bruynooghe, M., Denecker, M.: Analyzing semantics of aggregate answer set programming using approximation fixpoint theory. Computing Research Repository **cs.AI**(arXiv:2104.14789) (2021), `https://arxiv.org/abs/2104.14789`
68. Wang, Q., Zhang, Y., Wang, H., Geng, L., Lee, R., Zhang, X., Yu, G.: Automating incremental and asynchronous evaluation for recursive aggregate data processing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of data. pp. 2439–2454 (2020)
69. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)
70. Zaniolo, C., Arni, N., Ong, K.: Negation and aggregates in recursive rules: The LDL++ approach. In: International Conference on Deductive and Object-Oriented Databases. pp. 204–221. Springer (1993)
71. Zaniolo, C., Das, A., Gu, J., Li, Y., Li, M., Wang, J.: Monotonic properties of completed aggregates in recursive queries. Computing Research Repository **cs.DB**(arXiv:1910.08888) (2019), `http://arxiv.org/abs/1910.08888`
72. Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T., Interlandi, M.: Fixpoint semantics and optimization of recursive datalog programs with aggregates. Theory and Practice of Logic Programming **17**(5-6), 1048–1065 (2017)
73. Zhang, Y., Rayatidamavandi, M.: A characterization of the semantics of logic programs with aggregates. In: Proceedings of the International Joint Conference on Artificial Intelligence. pp. 1338–1344 (2016)