Recursive Rules with Aggregation: A Simple Unified Semantics (Extended Abstract)

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

Complex reasoning problems are most clearly and easily specified using logical rules, especially 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 different conflicting semantics.

This extended abstract gives an overview of 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 straightforwardly using their simple usual meaning.

Many computation problems, including complex reasoning problems in particular, are most clearly and easily specified using logical rules. 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, even the meaning of such rules has been challenging and remains a subject with significant complication and disagreement by experts.

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 clearly 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 a recursive rule, the semantics can be subtle.

Semantics of recursive rules with aggregation has been studied continuously since about 30 years ago, and intensively in the last several years, especially as they are needed in graph analysis and machine learning applications. However, the different semantics proposed, e.g., [7, 2], are complex and tricky, including having some experts changing their own minds about the desired semantics, e.g., [1, 2]. With such complex semantics, aggregation would be too challenging for non-experts to use correctly.

This extended abstract gives an overview of a simple unified semantics for recursive rules with aggregation, as well as negation and quantification, as described in the full paper [6]. The key idea is to support simple expression of the different assumptions underlying different semantics, and orthogonally interpret aggregation operations straightforwardly using their simple usual meaning. The unified 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 [3, 4], which has been used in a new language to support the power and ease of programming with logical constraints [5].

We applied the unified semantics to a variety of different examples, as described in detail in the full paper [6], including the longest and most sophisticated ones from dozens of previously studied examples [2]. For those from previously studied examples, instead of computing answer sets using naive guesses followed by sophisticated reducts, all of the results can be computed with a simple default assumption and a simple least fixed-point computation, as is used for formal inductive definitions and for commonsense reasoning. In all cases, we show that the resulting semantics match the desired semantics.

Overview of the Unified Semantics. Our simple and unified semantics for rules with aggregation as well as negation and quantification builds on founded semantics and constraint semantics [3, 4] for rules with negation and quantification. The founded semantics gives a single 3-valued model (i.e., the possible truth values of an assertion are true, false, and undefined) from simply a least fixed-point computation, and the constraint semantics gives a set of 2-valued models (i.e., the possible truth values of an assertion are true and false) from simply constraint solving.

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.

- First, if there is no aggregation or no potential non-monotonicity—that is, adding new facts used in the hypotheses of a rule may make the conclusion of a rule from true to false—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 example of Tom attending the logic seminar, 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.

- 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 as constraints, yielding a set of possible satisfying combinations. This is the constraint semantics.

- 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 example of Tom attending the logic seminar, the completion rule essentially says: 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 that Tom will attend, and one that Tom will not attend.

- Finally, an uncertain complete predicate can be further declared "closed", meaning that an assertion of the predicate is made false if inferring it to be true requires itself to be true.

  For the example of Tom attending the logic seminar, 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.

Overview of Language and Formal Semantics. Formal definitions of the language and semantics, and proofs of consistency and correctness of the semantics, appear in the full paper [6]. We give a brief overview here.

Our language supports Datalog rules extended with unrestricted negation, universal and existential quantification, aggregations, and comparisons. An *aggregation* has the form *agg S*, where *agg* is an aggregation operator (`count`, `min`, `max`, or `sum`), and S is a set expression of the form $\{X_1,\ldots,X_a : B\}$, where B is a conjunction of assertions or negated assertions. A *comparison* is an equality ($=$) or inequality ($\neq$, $\leq$, $<$, $\geq$, or $>$), with an aggregation on the left and a variable or constant on the right. Additional aggregation and comparison functions, including summing only the first component of a set of tuples and using orders on characters and strings, can be supported in the same principled way as we support those above. A *program* is a set of rules, facts, and declarations.

The key idea for extending the semantics is to identify conditions under which a comparison is definitely true or false in a 3-valued interpretation for the predicates, and to leave the comparison's truth value as undefined if those conditions don't hold. For example, consider the comparison $\texttt{count}\ \{X : \texttt{p}(X)\} \leq k$ and its complement (i.e., its negation) $\texttt{count}\ \{X : \texttt{p}(X)\} > k$. The former is true in an interpretation I if the number of ground instances of $\texttt{p}(X)$ that are true or undefined in I is at most k. The latter is true in I if the number of ground instances of $\texttt{p}(X)$ that are true in I is greater than k. Considering the complement eliminates the need to explicitly define when a comparison is false or undefined. Instead, we derive those conditions as follows: a comparison is false in I if its complement is true in I, and a comparison is undefined in I if neither it nor its complement is true in I.

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

References

[1] Michael Gelfond (2002): *Representing Knowledge in A-Prolog*. In: *Computational Logic: Logic Programming and Beyond*, Springer, pp. 413–451, doi:10.1007/3-540-45632-5_16.

[2] Michael Gelfond & Yuanlin Zhang (2019): *Vicious Circle Principle, Aggregates, and Formation of Sets in ASP Based Languages*. *Artificial Intelligence* 275, pp. 28–77, doi:10.1016/j.artint.2019.04.004.

[3] Yanhong A. Liu & Scott D. Stoller (2018): *Founded Semantics and Constraint Semantics of Logic Rules*. In: *Proceedings of the International Symposium on Logical Foundations of Computer Science (LFCS 2018)*, *Lecture Notes in Computer Science* 10703, Springer, pp. 221–241, doi:10.1007/978-3-319-72056-2_14.

[4] Yanhong A. Liu & Scott D. Stoller (2020): *Founded Semantics and Constraint Semantics of Logic Rules*. *Journal of Logic and Computation* 30(8). To appear. Preprint available at `https://arxiv.org/abs/1606.06269`.

[5] Yanhong A. Liu & Scott D. Stoller (2020): *Knowledge of Uncertain Worlds: Programming with Logical Constraints*. In: *Proceedings of the International Symposium on Logical Foundations of Computer Science (LFCS 2020)*, *Lecture Notes in Computer Science* 11972, Springer, pp. 111–127, doi:10.1007/978-3-030-36755-8_8. Also `https://arxiv.org/abs/1910.10346`.

[6] Yanhong A. Liu & Scott D. Stoller (2020): *Recursive Rules with Aggregation: A Simple Unified Semantics*. *Computing Research Repository* arXiv:2007.13053 [cs.DB]. `http://arxiv.org/abs/2007.13053`.

[7] Allen Van Gelder (1992): *The Well-Founded Semantics of Aggregation*. In: *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 127–138, doi:10.1145/137097.137854.