A Monotonic Extension for Horn-Clauses and its Significance in Datalog's Renaissance

Mirjana Mazuran[1], Edoardo Serra[2], and Carlo Zaniolo[3]

[1] Politecnico di Milano DEI – mazuran@elet.polimi.it
[2] University of Calabria DEIS – eserra@deis.unical.it
[3] University of California, Los Angeles – zaniolo@cs.ucla.edu

Abstract. FS-rules provide a powerful monotonic extension for Horn clauses that supports monotonic aggregates in recursion by reasoning on the multiplicity of occurrences satisfying existential goals. The least fixpoint semantics, and its equivalent least model semantics, hold for logic programs with FS-rules; moreover, generalized notions of stratification and stable models are easily derived once negated goals are also allowed. Finally, the generalization of techniques such as seminaive fixpoint and magic sets, make possible the efficient implementation of Datalog^{FS}, i.e., Datalog with FS-rules and stratified negation. A large number of applications that could not be supported efficiently, or could not be expressed at all in stratified Datalog can now be easily expressed and efficiently supported in Datalog^{FS} and a powerful Datalog^{FS} system is now being developed at UCLA.

1 Introduction

The recent revival of interest in Datalog [1] is driven by various developments that include the emergence of natural application areas [2–4], the success of industrial-strength systems [5], and Datalog's uses in (i) advanced computational and semantic models [3, 6], (ii) the big-data problem [7, 8], and (iii) Data Stream Management Systems [9]. Due to space limitations, this is a very incomplete list, which does not mention many significant contributions from the past, and the many new ones that are emerging now, i.e., in a time that has been described with terms such as 'resurgence' [1], 'springtime' [3] and 'renaissance' [10] for Datalog[4]. In this paper, we make a significant contribution to this renaissance, by providing an effective solution to the problem of supporting aggregates in recursive rules, a challenge that had motivated much classical Datalog research [11–16]. Space constraints force us to limit this presentation to a general overview, whereas details and formal proofs are given in [17, 18].

2 A Monotonic Extension for Horn Clauses

There is a big party on campus, and every student who has a friend attending the party will join in. This can be expressed by the following rule:

[4] The last term is actually the most fitting, since the renaissance is the era that, after the 'dark ages,' revived arts and sciences producing accomplishments that outshined and outlasted even the glorious ones of classical times.

$$\texttt{attend(Y)} \leftarrow \texttt{student(Y)}, \texttt{attend(X)}, \texttt{friend(Y, X)}.$$

A logical equivalent that makes a distinction between universal and existential variables is:

$$\forall \texttt{Y}[\texttt{attend(Y)} \leftarrow \texttt{student(Y)}, \exists \texttt{X}[\texttt{attend(X)}, \texttt{friend(Y, X)}]]$$

which shows that `Y` is a global/universal variable and `X` is a local/existential one. Now, if the party is held during finals, students are much less outgoing and require that three friends attend the party before they join in. We could express this condition by expanding the bracketed expression above into: $[\texttt{attend(X)}, \texttt{friend(X, Y1)}, \texttt{friend(X, Y2)}, \texttt{friend(X, Y3)}, \texttt{Y1}\neq\texttt{Y2}, \texttt{Y2}\neq\texttt{Y3}, \texttt{Y3}\neq\texttt{Y1}]$. However, such an expansion becomes unwieldy when the number of required friends increases, and actually impossible when this number is a variable. Thus, Datalog^{FS} introduces a special notation as follows:

$$\texttt{attend(Y)} \leftarrow \texttt{student(Y)}, \texttt{3:[attend(X)}, \texttt{friend(Y, X)]}.$$

Here, $\texttt{3:[attend(X)}, \texttt{friend(Y, X)]}$ means that there are at least three distinct occurrences of the local variable `X` that make the expression in the brackets true. In general , $\texttt{K:[b-expression(X, Y)]}$, where `X` is the vector of global variables and `X` is the vector of local variables, means that there are at least `K` distinct occurrences of `Y` that satisfy our $\texttt{b-expression(X, Y)}$. Naturally, if $\texttt{K:[b-expression(X, Y)]}$ is true, then $\texttt{K}' : \texttt{[b-expression(X, Y)]}$ is also true for every $1 \leq \texttt{K}' \leq \texttt{K}$. Following [17], we refer to the conjunct of positive atoms in the bracket as the *b-expression*, while the whole '$\texttt{3:[attend(X)}, \texttt{friend(Y, X)]}$' is called a *Running FS-goal*.

Rules where FS-goals are allowed will be called *FS-rules*. An FS-rule has the form $A \leftarrow A_1, \ldots, A_m$, where A is an atom, which is the *head* of the rule, and $A_1, \ldots, A_m$ is the conjunction of literals forming the *body* of the rule. Now, literals in the body can either be (i) the positive atom of Horn clauses, or (ii) running FS-goals. A set of FS-rules will be called an *FS-program*.

The elegant foundations that provide formal semantics to definite-clause programs find natural extensions, since the notions of Herbrand Universe, interpretations, models, instantiated rules and programs, and the immediate consequence operator T_P, can be extended to an FS-program [18]. The model intersection property holds for FS-programs, whereby every FS-program has a least minimal model. Moreover, the immediate consequence operator T_P of an FS-program P is monotonic and continuous in the lattice of interpretations, whereby the equation $I = T_P(I)$ has a least solution, denoted $lfp(T_P)$, which is equal to the least model of P. Finally, $lfp(T_P)$ can be computed as $lfp(T_P) = T_P^{\uparrow\omega}(\emptyset)$. These beautiful properties that Paris Kanellakis once described as 'The Garden of Eden' of declarative semantics[5] have now been extended to logic programs with FS-rules, which can now express declaratively many monotonic functions which were not expressible in traditional Datalog.

However, many real-life applications require non-monotonic functions and reasoning. Indeed, Datalog and its bottom-up semantics fostered many advances

[5] Paris Kanellakis, personal communication, March 1987.

in this area with the introduction of stratified negation, stable models, and related semantics. As shown in [18], these non-monotonic concepts and definitions can be easily extended to FS-programs. For an example, let us return to the party attendance example inspired by [15] and add a few facts describing students and their friends:

Example 1. Organizers always attend; the others join after three friends do.

$$\texttt{organizer(tom)}. \quad \texttt{organizer(sue)}. \quad \texttt{organizer(pat)}.$$

$$\begin{array}{lll} \texttt{friend(marc, sue)}. & \texttt{friend(marc, tom)}. & \texttt{friend(marc, pat)}. \\ \texttt{friend(ann, pat)}. & \texttt{friend(ann, tom)}. & \texttt{friend(ann, marc)}. \\ \texttt{student(marc)}. & \texttt{student(ann)}. & \end{array}$$

$$\begin{array}{l} \texttt{attend(Y)} \leftarrow \texttt{organizer(Y)}. \\ \texttt{attend(Y)} \leftarrow \texttt{student(Y)}, \texttt{3}:[\texttt{attend(X)}, \texttt{friend(Y, X)}]. \end{array}$$

In this example, `tom`, `pat` and `sue` attend the party as organizers. Now, `marc` views the three of them as his friends, so he will attend too. Because of this, `ann` who views `pat`, `tom` and `marc` as her friends, joins the party too.

Negation is needed to detect how many people actually attend the party:

$$\texttt{partycount(K)} \leftarrow \texttt{K}:[\texttt{attend(Y)}], \texttt{K1} = \texttt{K} + \texttt{1}, \ \neg\texttt{K1}:[\texttt{attend(Y1)}].$$

Thus, at least `K` people will attend the party but `K+1` will not. Therefore, `K` is the exact count of people attending the party. Alternatively the *final FS* construct, denoted by =! can be used to determine the exact count, as follows:

$$\texttt{partycount(K)} \leftarrow \texttt{K=!}\,[\texttt{attend(Y)}].$$

The meaning of this rule is actually defined by its expansion into the previous one that uses negation—whereby we refer to programs that are stratified w.r.t. final FS goals as negation-stratified programs.

3 Datalog^{FS}

In addition to stratified negation, the Datalog^{FS} system being developed at UCLA supports FS-assert constructs that are used to declare predicates with multiplicity greater than one. For instance, we might want to state that `tom` has five friends without stating their names as follows: `friends(tom):5`. Then, since `tom` has five friends, the following rule that invites to the party students with more than four friends will succeed for `tom`:

$$\texttt{invite(Y)} \leftarrow \texttt{student(Y)}, \texttt{4}:[\texttt{friends(Y)}].$$

This FS-assert construct is basically syntactic sugaring whose semantics is defined by a simple rewriting. In fact `friends(tom):5` is viewed as a shorthand for $\overline{\texttt{friends}}(\texttt{tom}, \texttt{J}), \texttt{J} = \texttt{1}, \ldots, \texttt{5}$. Naturally the FS-goals in the rules are re-written according to this expansion, whereas our previous rule becomes:

$$\texttt{invite(Y)} \leftarrow \texttt{student(Y)}, \texttt{4}:[\ \overline{\texttt{friends}}(\texttt{Y}, \texttt{J})].$$

The FS-assert construct is very useful since it implicitly computes the maximum of positive integers. For instance, say that we add the fact $\texttt{friends(tom)}:\texttt{7}$. Since this fact stands for $\overline{\texttt{friends}}(\texttt{tom}, \texttt{J}), \texttt{J} = \texttt{1}, \ldots, \texttt{7}$, it subsumes $\overline{\texttt{friends}}(\texttt{tom}, \texttt{J}), \texttt{J} = \texttt{1}, \ldots, \texttt{5}$, and therefore $\texttt{friends(tom)}:\texttt{5}$.

By using running FS-goals, and FS-assert constructs, negation-stratified Datalog^{FS} programs can express in a concise fashion queries that were not expressible in stratified Datalog. For instance, the Summarized Part-Explosion query cannot be expressed in Datalog with stratified aggregates [14]. This query counts the number of copies of component `Sub` needed to construct one copy of a given `Part`:

Example 2. Summarized Part Explosion.

$$\texttt{cassb(Part, Sub)}:\texttt{Qty} \leftarrow \texttt{subpart(Part, Sub, Qty)}.$$
$$\texttt{need(Sub, Sub)}:\texttt{1} \leftarrow \quad \texttt{subpart}(\_, \texttt{Sub}, \_).$$
$$\texttt{need(Part, Sub)}:\texttt{K} \leftarrow \texttt{K}:[\texttt{cassb(Part, P1)}, \texttt{need(P1, Sub)}].$$
$$\texttt{total(Part, Sub, K)} \leftarrow \texttt{K} =![\texttt{need(Part, Sub)}].$$

We next consider the Company Control application proposed in [11].

Example 3. Companies can purchase shares of other companies; in addition to its directly owned shares, a company A controls the shares controlled by a company B when A has a controlling majority (50%) of B's shares (in other words, when A bought B). The shares of each company are subdivided in 100 equal-size lots.

$$\texttt{cshares(C2, C3, dirct)}:\texttt{P} \leftarrow \quad \texttt{owned\_shares(C2, C3, P)}.$$
$$\texttt{cshares(C1, C3, indirct)}:\texttt{P} \leftarrow \texttt{P}:[\texttt{bought(C1, C2)}, \texttt{cshares(C2, C3, \_)}].$$
$$\texttt{bought(C1, C2)} \leftarrow \quad \texttt{C1} \neq \texttt{C2}, \quad \texttt{50}:[\texttt{cshares(C1, C2, \_)}].$$

Here `dirct` and `indirct` are tags identifying the two different kinds of shares.

Simple assemblies, such as bicycles, can be put together the very same day in which the last basic part arrives. Thus, the time needed to deliver a bicycle is the maximum of the number of days that the various basic parts require to arrive.

Example 4. How many days until delivery?

$$\texttt{delivery(Pno)}:\texttt{Days} \quad \leftarrow \quad \texttt{basic(Pno, Days)}.$$
$$\texttt{delivery(Part)}:\texttt{Days} \leftarrow \quad \texttt{assbl(Part, Sub, \_)}, \texttt{Days}:[\texttt{delivery(Sub)}].$$
$$\texttt{actualDays(Part, CDays)} \leftarrow \texttt{CDays} =![\texttt{delivery(Part)}].$$

For each assembled part, we find each basic subpart along with the number of days this takes to arrive. By using the multi-occurring predicate `delivery` inside the FS-goal '$\texttt{Days}:[\texttt{delivery(Sub)}]$' we find, for a given `Part`, the maximum among the delivery times of its subparts.

4 Efficient Implementation

Datalog^{FS} programs are amenable to efficient implementation using (i) generalizations of well-known techniques such as the seminaive (or differential) fixpoint and the magic set method, and (ii) a specialized new technique called *max optimization* that was introduced specifically for Datalog^{FS} [17] and is described

next. The max optimization of the rule in Example 1 begins by recasting it into this equivalent rule:

$$\texttt{attend(Y)} \leftarrow \texttt{student(Y)}, \texttt{K}{:}[\texttt{friend(Y, X)}, \texttt{attend(X)}], \texttt{K} \geq 3.$$

Here, the value of K ranges from 1 to a $max(\texttt{K})$, thus achieving monotonicity and least-fixpoint semantics. However, we also observe that the rule is actually satisfied iff $max(\texttt{K}) \geq 3$. In other words, we do not need to compute a continuous count, we can instead perform a final count computation at each iteration in the seminaive fixpoint computation. The same conclusion holds for all the examples in this paper, and in fact for all the rules that use only monotonic functions on positive numbers. In these programs, final FS-goals can be computed by ignoring every K value but $max(\texttt{K})$. Therefore, traditional count and sum can be used to implement these rules, instead of continuous aggregates. Indeed, the max-based optimization can be performed whenever the function that maps FS-values from the body to the head is monotonic on positive numbers. This is true for all examples in this paper where the mapping is the identity function, but monotic arithmetic functions such as addition, multiplication and many other functions easily recognized as monotonic by the compiler can be used [17].

As discussed in [17], the standard differential fixpoint and magic-set transformations can be applied to FS-rules, but only after they have been put in *canonical form*. Rules with FS-goals can be reduced into canonical form by simply moving the predicates in the b-expression out of the brackets while avoiding redundancy. For instance for Example 1 we obtain the following equivalent rule:

$$\begin{aligned}\texttt{attend(Y)} \leftarrow\ & \texttt{student(Y)}, \texttt{friend(Y, X)}, \texttt{attend(X)},\\ & \texttt{K}{:}[\texttt{friend(Y, XX)}, \texttt{attend(XX)}], \texttt{K} \geq 3.\end{aligned}$$

Then, seminaive fixpoint will be computed by performing a symbolic differentiation on the recursive predicates outside the brackets whereas b-expressions are left unchanged, as if they were constants:

$$\begin{aligned}\delta\texttt{attend(Y)} \leftarrow\ & \texttt{student(Y)}, \texttt{friend(Y, X)}, \delta\texttt{attend(X)},\\ & \texttt{K}{:}[\texttt{friend(Y, XX)}, \texttt{attend(XX)}], \texttt{K} \geq 3.\end{aligned}$$

The canonical representation is used when performing the binding passing analysis and in the magic-set method that propagate constraints in a top-down fashion. For instance, say that in Example 1 we want to know whether a given Joe will attend, using the goal: ?attend($Joe). Then, after performing the binding passing analysis, we apply the magic-set transformation and obtain the following magic set rules, where the b-expression condition has also been relaxed:

$$\begin{aligned}&\texttt{m.attend(\$Joe)}.\\ &\texttt{m.attend(X)} \leftarrow\ \texttt{m.attend(Y)}, \texttt{student(Y)}, \texttt{friend(Y, X)},\\ &\qquad\qquad\qquad\quad \texttt{K}{:}[\texttt{friend(Y, \_)}], \texttt{K} \geq 3.\end{aligned}$$

Therefore, the magic set consists of Joe's friends and the friends of his friends (friend*); but if Joe has fewer than three friends, we can exclude him from the magic set—and the same holds for any friend*. Once the magic set predicate is computed as shown above, m.attend(Y) is added as a goal to the original exit rules and recursive rules restricting the final seminaive fixpoint computation [18].

5 Arbitrary Positve Numbers

The least fixpoint and its equivalent least model define the semantics of Datalog^{FS} when the FS-values are positive integers, and this also provides a declarative semantics for programs that use arbitrary positive numbers for FS-values. In fact, the rational numbers in a program can be represented by their numerators once we assume they all share the same (large) denominator, D. Then operations on these numbers can be viewed as involving only their numerators: e.g., $A/D + B/D = A + B/D$, and $A/D \times B/D = ((A \times B) \div D)/D$. Now, while addition introduces no error, the multiplication introduces a roundoff error due to integer division $\div D$. However, roundoff is an arithmetic function that is monotonically increasing (it can be viewed as staircase), and thus our Datalog^{FS} programs still have a least fixpoint-based semantics. Now, since large values for D would produce good approximations, rather than unbounded length integers, we can instead use floating point numbers to provide accurate solutions efficiently supported in systems [17]. Because of limited precision mantissa, floating point numbers also incur in roundoff errors; but again, these are monotonic functions, and any imprecision in the resulting fixpoint can be resolved with double-precision arithmetic and various methods of numerical analysis. Finally, observe that monotonic approximation preserves the max-based optimization in the computation—a sine qua non since the numerators are now large integers.

Many important applications that use probabilities and fractional weights can now be expressed concisely and supported efficiently. Examples include shortest-path in graphs, page rank, social networks [17], and the following example.

Example 5. Say that arc(a, b):0.66 denotes that starting from a we reach b in 66% of cases. Then, the following program computes the probability of completing a trip from a to Y along the maximum-probability path:

```
reach(a):1.00.
reach(Y):V ←   reach(X), V:[reach(X), arc(X, Y)].
maxprob(Y, V) ← V =![reach(Y)].
```

The source a is reachable with probability 1. Then, the probability of reaching Y via an arc from X is the product of the probability of being in X times the probability that the segment from X to Y can be completed. This product is computed by the goal V : [reach(X), arc(X, Y)] in the first rule. Finally, in the head of the last rule, we only retain the maximum V—i.e., we only retain the path with largest probability to succeed. □

6 Conclusion

FS-rules provide a simple but powerful extension of Horn Clauses which dovetails with both the declarative semantics of Datalog and its bottom-up implementation technology. In fact, the inclusion of running FS-goals, which operate in ways that are similar to continuous counts, produces a generalized T_P operator that is monotonic and continuous in the lattice of interpretations: thus its repeated application, starting with an empty interpretation, converges (on or before the

first ordinal) to T_P's least fixpoint, which coincides with the unique minimal model for P. Moreover, the bottom-up optimization techniques of Datalog, such as magic-sets and differential fixpoints, can be easily generalized to programs with FS-rules; simple generalizations also hold for stratified negation and more advanced non-monotonic semantics [18]. Applications that cannot be expressed efficiently, or cannot be expressed at all, in Datalog can be expressed efficiently in the powerful Datalog^{FS} system being developed at UCLA. Inasmuch as Datalog's recursive query techniques greatly influenced their SQL implementations [19, 20], these extensions can also lead to their support in commercial DBMS.

References

1. Pablo Barceló and Reinhard Pichler, editors. *Datalog in Academia and Industry–2nd International Workshop, Datalog 2.0*, volume 7494 of *LNCS*. Springer, 2012.
2. Boon Thau Loo et al. Declarative networking. *Commun. ACM*, 52(11), 2009.
3. Joseph M. Hellerstein. Datalog redux: experience and conjecture. In *PODS*, pages 1–2, 2010.
4. Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Emilien Antoine. A rule-based language for web data management. In *PODS*, pages 293–304, 2011.
5. Todd J. Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In Barceló and Pichler [1], pages 1–8.
6. Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, pages 2–13, 2011.
7. Foto N. Afrati et al. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8, 2011.
8. Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.
9. Carlo Zaniolo. The logic of query languages for data streams. In *Logic and Databases 2011. EDBT 2011 Workshops*, pages 1–2, 2011.
10. Serge Abiteboul. Datalog: La renaissance.
11. Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.
12. Phokion G. Kolaitis. The expressive power of stratified logic programs. *Inf. Comput.*, 90:50–66, January 1991.
13. Sergio Greco and Carlo Zaniolo. Greedy algorithms in datalog. *TPLP*, 1(4):381–407, 2001.
14. Inderpal Singh Mumick and Oded Shmueli. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence*, 15:407–435, 1995.
15. Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive database. *J. Comput. Syst. Sci.*, 54(1):79–97, 1997.
16. C. Zaniolo et al. *Advanced Database Systems*. Morgan Kaufmann, 1997.
17. Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *The VLDB Journal On-Line First, Nov. 2012*.
18. Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. A declarative extension of horn clauses, and its significance for datalog and its applications. February 2013.
19. Inderpal Singh Mumick and Hamid Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD Conference*, pages 103–114, 1994.
20. Carlos Ordonez. Optimization of linear recursive queries in sql. *IEEE Trans. Knowl. Data Eng.*, 22(2):264–277, 2010.