Online appendix for the paper

A Declarative Extension of Horn Clauses, and its Significance for Datalog and its Applications

published in Theory and Practice of Logic Programming

Mirjana Mazuran
Politecnico di Milano DEIB

Edoardo Serra
University of Maryland

Carlo Zaniolo
University of California, Los Angeles

submitted 10 April 2013; revised 23 May 2013; accepted 23 June 2013

Appendix A Syntax of Datalog^{FS}

We can now summarize the syntax of our Datalog^{FS} programs obtained by enriching Horn clauses with the folllowing constructs:

- Running-FS goals
- Multi-occurring predicates, and
- Final-FS goals.

A Datalog^{FS} program is a set of rules where the head literal can either be an atom or an FS-assert statement. An FS-assert statement describes multi-occurring predicates and is an atom followed by `:K`, where `K` is either a constant or a variable. The literals in the body of Datalog^{FS} rules can be either atoms or negated atoms, as in Datalog, or they can be *FS-goals*. Now FS-goals come in the following two forms:

- `Kj:[exprj]` for a *Running-FS goal,* and
- `Kj =![exprj]` for a *Final-FS goal.*

where `exprj` is a conjunction of positive atoms, and `Kj` (which is called the FS-term) can either be constant or a variable not contained in `exprj`. We require our programs to be stratified w.r.t negation and Final FS-goals.

We will now illustrate the many interesting applications of our stratified Datalog^{FS} programs, and then discuss the technology for their efficient implementation.

Appendix B Formal Properties

Proof of Theorem 1: *If M_1 and M_2 are two models of an FS-program P, their intersection $M_1 \cap M_2$ is also a model for P.*
To proof this theorem, we need to show that every rule $r \in ground(P)$ is satisfied in

$M_1 \cap M_2$. In fact, let $h(r)$ be the head of our rule: if $h(r)$ is in both M_1 and M_2 then it is also in $M_1 \cap M_2$, which will thus satisfy the rule. Otherwise say that $h(r)$ is not in M_1: then the body of r cannot be satisfied by M_1, and thus cannot be satisfied by $M_1 \cap M_2$ which is a subset of M_1. Symmetrically for M_2. QED.

Proof of Theorem 2: *Let P be an FS-program with immediate consequence operator T_P and least model $M(P)$. Then $lfp(T_P) = M(P)$.*
To proof this, let I be a fixpoint for T_P. Now, if I is not a model for P, then $ground(P)$ must contain a rule r s.t. the body of r is satisfied by I but $h(r)$ is not in I; then I is not a fixpoint—a contradiction. Therefore, the least fixpoint $lfp(T_P)$ must be a model for P. Now, for each atom a in the least model M_P, $ground(P)$ must contain some rule r, such that a is the head of r and the body of r is satisfied in M_P (otherwise $M_P - \{a\}$ would still be a model, a contradiction). Therefore, $T_P(M_P) \supseteq M_P$. But if $T_P(M_P) \supset M_P$, then M_P cannot be a model. Therefore, M_P is a fixpoint, whereby $M_P \supseteq lfp(T_P)$. But since every fixpoint is a model we also have that $lfp(T_P) \supseteq M_P$. QED
Proof-Theoretic Semantics We now sketch a way in which the semantics of running FS-goals can be expressed with standard Horn clauses, and thus supported using SLD-resolution.

Let `Global` and `Local` respectively denote the global (i.e., universal) and local (i.e., existential) variables in our b-expression which is a conjunct of one or more positive atoms:

$$..., \mathtt{K} : [\mathtt{bexpr}(\mathtt{Global}, \mathtt{Local})], ...$$

Let us then use the following rule to represent `bexpr` into a single predicate where `GL` and `LL`, respectively denote the list of the global variables and local variables:

$$\mathtt{bgoal}(\mathtt{GL}, \mathtt{LL}) \leftarrow \quad \mathtt{bexpr}(\mathtt{Global}, \mathtt{Local}).$$

Now let us express $\mathtt{s}(\mathtt{N}) : [\mathtt{bgoal}(\mathtt{GL}, \mathtt{LL})]$ by the predicate $\mathtt{fscnt}(\mathtt{GL}, \mathtt{LL}, \mathtt{s}(\mathtt{N}))$ which can be defined by the following rules:

$$\begin{array}{ll}
\mathtt{fscnt}(\mathtt{GL}, [\mathtt{LL}], 1) \leftarrow & \mathtt{bgoal}(\mathtt{GL}, \mathtt{LL}). \\
\mathtt{fscnt}(\mathtt{GL}, [\mathtt{LL}|\mathtt{Bag}], \mathtt{s}(\mathtt{K})) \leftarrow & \mathtt{fscnt}(\mathtt{GL}, \mathtt{Bag}, \mathtt{K}), \mathtt{bgoal}(\mathtt{GL}, \mathtt{LL}), \mathtt{notin}(\mathtt{LL}, \mathtt{Bag}). \\
\mathtt{notin}(\mathtt{LL}, [\,]). & \\
\mathtt{notin}(\mathtt{LL}, [\mathtt{H}|\mathtt{Rest}]) \leftarrow & \mathtt{neq}(\mathtt{LL}, \mathtt{H}), \mathtt{notin}(\mathtt{LL}, \mathtt{Rest}).
\end{array}$$

where `neq` is a built-in monotone predicate that is true if its second argument is different from its first one. Thus our `fscnt`-based rewriting re-expresses the abstract semantics of the running fs-count construct of Datalog^{FS}.

While this is not an issue in terms of abstract SLD semantics which ignores non-terminating derivations, the previous rules can be modified into the following terminating Prolog program with tail-recursion.

$$\begin{array}{ll}
\mathtt{fscntl}(\mathtt{GLl}, \mathtt{K}) \leftarrow & \mathtt{fscntl1}(\mathtt{GL}, [], 0, \mathtt{K}). \\
\mathtt{fscntl1}(\mathtt{GL}, \mathtt{Bag}, \mathtt{K}, \mathtt{s}(\mathtt{K}) \leftarrow & \mathtt{kexpr}(\mathtt{GL}, \mathtt{LL}), \mathtt{notin}(\mathtt{LL}, \mathtt{Bag}). \\
\mathtt{fscntl1}(\mathtt{GL}, \mathtt{Bag}, \mathtt{K}, \mathtt{K1}) \leftarrow & \mathtt{kexpr}(\mathtt{GL}, \mathtt{LL}), \mathtt{notin}(\mathtt{LL}, \mathtt{Bag}), \\
 & \mathtt{fscntls1}(\mathtt{GL}, [\mathtt{LL}|\mathtt{Bag}], \mathtt{s}(\mathtt{K}), \mathtt{K1}).
\end{array}$$

Appendix C Efficient Implementation

The three main techniques that make possible very efficient implementations for Datalog^{FS} can be listed in the order in which they are applied by the compiler as follows (i) the magic-set transformation, (i) differential (a.k.a. semi-naive) fixpoint, and (iii) the max-based optimization. The max-based optimization has been reported in Section 6 of the paper, and the extension of differential fixpoint and magic set to Datalog^{FS} is reported below. The differential fixpoint technique, which was already presented in (Mazuran et al. 2012), is summarized below for the convenience of the reader. However, the magic-set technique for Datalog^{FS} is new and not published in previous papers.

Differential Fixpoint

The differential fixpoint (a.k.a., the seminaive fixpoint) method, which represents the cornerstone of the bottom-up implementation for Datalog programs (Arni et al. 2003; Zaniolo et al. 1997), is also applicable and effective for Datalog^{FS} programs. The method applies a *reduced differentiation step* upon recursive FS-rules that are in *canonical form*. Moreover every rule can put into canonical form by rewriting its FS-goals by (i) a relaxed factorization step, and (ii) a reduction step (Mazuran et al. 2012).

Relaxed Factorization This transformation replaces an FS goal $\texttt{K}:[\texttt{expr}(\texttt{X},\texttt{Y})]$ (where X and Y denote the global and local variables) by the equivalent pair of goals:

$$\texttt{expr}(\texttt{X},\_),\texttt{K}:[\texttt{expr}(\texttt{X},\texttt{Y}_2)]$$

Take for instance the following recursive rule:

$$\texttt{reach}(\texttt{Y}):\texttt{V} \leftarrow \texttt{reach}(\texttt{X}),\ \texttt{V}:[\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y})].$$

where X and Y are global, and there are no local variables in the b-expression of this rule; thus the expansion step produces:

$$\texttt{reach}(\texttt{Y}):\texttt{V} \leftarrow \texttt{reach}(\texttt{X}),\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y}),\texttt{V}:[\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y})].$$

Reduction Step: This step removes each redundant goal introduced by the previous step. For the example at hand, we see that reach(X) appears twice, and thus one of the occurrences can be eliminated, producing:

$$\texttt{reach}(\texttt{Y}):\texttt{V} \leftarrow \texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y}),\texttt{V}:[\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y})].$$

In general the reduction step will merge pairs of goals that are identical except for a renaming of variables. The applications of these two steps reduces the rules to *canonical form*. We can now proceed with the reduced differentiation of the canonical rules.

Reduced Differentiation: The rules produced by the reduction step are differentiated with their *b-expression treated as constants* (i.e., as if the predicates in the brackets were not recursive or mutually recursive with the head of the rule). Thus, for the example at hand, we obtain:

$$\delta\texttt{reach}(\texttt{Y}):\texttt{V} \leftarrow \delta\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y}),\texttt{V}:[\texttt{reach}(\texttt{X}),\texttt{arc}(\texttt{X},\texttt{Y})].$$

More examples and details can be found in (Mazuran et al. 2012). Here we consider a second version of Floyd's Algorithm.

Another Version of Floyd's Algorithm and its Implementation

Consider a weighted graph represented by $\texttt{arc}(\texttt{X},\texttt{Y},\texttt{D})$ where $\langle X, Y\rangle$ is the edge and D is its weight. Assume that `upperb` is the sum of the weights of all edges in the graph added by 1. We implement the Floyd algorithm with the following Datalog^{FS} program:

$$\begin{array}{ll}
\texttt{fpath}(\texttt{X},\texttt{Y}):\texttt{K} \leftarrow & \texttt{arc}(\texttt{X},\texttt{Y},\texttt{D}), \texttt{K} = \texttt{upperb} - \texttt{D}.\\
\texttt{fpath}(\texttt{X},\texttt{Z}):\texttt{K} \leftarrow & \texttt{node}(\texttt{Y}), \texttt{K1} : [\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{K2} : [\texttt{fpath}(\texttt{Y},\texttt{Z})],\\
& \texttt{K} = \texttt{K1} + \texttt{K2} - \texttt{upperb}.\\
\texttt{sp}(\texttt{X},\texttt{Z},\texttt{D}) \leftarrow & \texttt{K!} = [\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{D} = \texttt{upperb} - \texttt{K}.
\end{array}$$

That is, the first rule scales the weights of the edges in the graph with respect to the upper bound. The second rule computes the maximum weight of a path from X to Z again scaled with respect to the upper bound, that is, we have to sum the weight of the two paths: (i) from X to Y and (ii) from Y to Z. The weight of (i) is $upperb - K1$ and the weight of (ii) is $upperb - K2$, thus by summing the two and then scaling we have $D = upperb - [(upperb - K1) + (upperb - K2)] = K1 + K2 - upperb$. Finally, the third rule scales this value for the last time and we obtain a weight that corresponds to the minimum one.

Consider the graph represented by the following arc facts:

$$\{\texttt{arc}(\texttt{a},\texttt{b},1), \texttt{arc}(\texttt{a},\texttt{c},5), \texttt{arc}(\texttt{b},\texttt{c},2), \texttt{arc}(\texttt{c},\texttt{d},3)\}$$

then we have $upperb = 11$. The least fixpoint execution is:

$$\begin{array}{ll}
\texttt{I}_0 = & \{\texttt{fpath}(\texttt{a},\texttt{b}) : 10, \texttt{fpath}(\texttt{a},\texttt{c}) : 6, \texttt{fpath}(\texttt{b},\texttt{c}) : 9, \texttt{fpath}(\texttt{c},\texttt{d}) : 8\}\\
\texttt{I}_1 = & \{\texttt{fpath}(\texttt{a},\texttt{b}) : 10, \texttt{path}(\texttt{a},\texttt{c}) : 8, \texttt{fpath}(\texttt{b},\texttt{c}) : 9, \texttt{fpath}(\texttt{c},\texttt{d}) : 8, \texttt{fpath}(\texttt{a},\texttt{d}) : 3,\\
& \texttt{fpath}(\texttt{b},\texttt{d}) : 6\}\\
\texttt{I}_2 = & \{\texttt{fpath}(\texttt{a},\texttt{b}) : 10, \texttt{path}(\texttt{a},\texttt{c}) : 8, \texttt{fpath}(\texttt{b},\texttt{c}) : 9, \texttt{fpath}(\texttt{c},\texttt{d}) : 8, \texttt{fpath}(\texttt{a},\texttt{d}) : 5,\\
& \texttt{fpath}(\texttt{b},\texttt{d}) : 6\}
\end{array}$$

$$\texttt{I}_3 = \{\texttt{sp}(\texttt{a},\texttt{b},1), \texttt{sp}(\texttt{a},\texttt{c},3), \texttt{sp}(\texttt{b},\texttt{c},2), \texttt{sp}(\texttt{c},\texttt{d},3), \texttt{sp}(\texttt{a},\texttt{d},6), \texttt{sp}(\texttt{b},\texttt{d},5)\}$$

Now, consider the following rule from the program:

$$\begin{array}{ll}
\texttt{fpath}(\texttt{X},\texttt{Z}):\texttt{K} \leftarrow & \texttt{node}(\texttt{Y}), \texttt{K1} : [\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{K2} : [\texttt{fpath}(\texttt{Y},\texttt{Z})],\\
& \texttt{K} = \texttt{K1} + \texttt{K2} - \texttt{upperb}.
\end{array}$$

The relaxed factorization yields:

$$\begin{array}{ll}
\texttt{fpath}(\texttt{X},\texttt{Z}):\texttt{K} \leftarrow & \texttt{node}(\texttt{Y}), \texttt{fpath}(\texttt{X},\texttt{Y}), \texttt{fpath}(\texttt{Y},\texttt{Z}),\\
& \texttt{K1}:[\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{K2}:[\texttt{fpath}(\texttt{Y},\texttt{Z})],\\
& \texttt{K} = \texttt{K1} + \texttt{K2} - \texttt{upperb}.
\end{array}$$

No redundant goals can be eliminated here, whereby we move the standard differentiation that yields:

$$
\begin{aligned}
\delta\texttt{fpath}(\texttt{X},\texttt{Z}):\texttt{K} \leftarrow\ & \texttt{node}(\texttt{Y}), \delta\texttt{fpath}(\texttt{X},\texttt{Y}), \texttt{fpath}(\texttt{Y},\texttt{Z}),\\
& \texttt{K1}:[\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{K2}:[\texttt{fpath}(\texttt{Y},\texttt{Z})],\\
& \texttt{K} = \texttt{K1} + \texttt{K2} - \texttt{upperb}.\\
\delta\texttt{fpath}(\texttt{X},\texttt{Z}):\texttt{K} \leftarrow\ & \texttt{node}(\texttt{Y}), \texttt{fpath}(\texttt{X},\texttt{Y}), \delta\texttt{fpath}(\texttt{Y},\texttt{Z}),\\
& \texttt{K1}:[\texttt{fpath}(\texttt{X},\texttt{Y})], \texttt{K2}:[\texttt{fpath}(\texttt{Y},\texttt{Z})],\\
& \texttt{K} = \texttt{K1} + \texttt{K2} - \texttt{upperb}.
\end{aligned}
$$

Thus, we have now reduced the differential fixpoint optimization for Datalog^{FS} to that of standard Datalog, whereby we can use its well-understood and widely tested optimization techniques. For instance, a further optimization that is supported by many Datalog compilers avoids having to repeat the computation of $\delta\texttt{fpath}(\texttt{X},\texttt{Y}), \delta\texttt{fpath}(\texttt{Y},\texttt{Z})$ in both rules. This improvement is thus applicable to Datalog^{FS} as well.

More examples and details can be found in (Mazuran et al. 2012).

Differential Fixpoint, Aggregates and Greedy Algorithms

The differential fixpoint implementation of many algorithms that are similar to those we have discussed here recently appeared in (Seo et al. 2013). The reference paper shows that performance levels approaching those of procedural programs can be achieved via Datalog-like rules that use monotonic aggregates in recursive computations. The paper builds on operational semantics, rather than the declarative semantics, and this does not allow a clear connection with theoretical concepts, such as model-theoretic semantics or stable models, which instead provide formal foundations for our paper. In our approach, we instead start from declarative semantics, and turn it into efficient implementation via (i) Max-based optimization, and (ii) differential fixpoint. Every program we obtain from these two optimization steps can be supported using the efficient implementation techniques described in (Seo et al. 2013). However, the inverse is not true and in particular we conjecture that the Dijkstra's algorithm discussed in (Seo et al. 2013) cannot be expressed in Datalog^{FS}. Indeed, if we consider the maximum probability path problem, Example 9 in the main paper, and we solve it using Dijkstra's algorithm, we must use the arcs departing from the node which has currently the highest probability. Now, the computation of such a node is not monotonic. (Indeed, if we enlarge a set of pairs $(\texttt{person},\texttt{age})$, the max age in the set behaves monotonically, but not the person who is the oldest.) The symmetric argument applies to the shortest path, and it is likely to apply to greedy algorithms in general. A second issue is that the traditional computation of the differential fixpoint for linear rules only uses the delta values produced at the last step. However in a greedy algorithm, the determination of the max or min will have to take into consideration the values of all the nodes considered so far, and not yet used. Thus, general compilation/optimization techniques starting from operational semantics might be harder to derive than starting from declarative semantics, as it is in fact done for Datalog^{FS} in the implementation described in (Shkapsky et al. 2013).

The Magic-Set Method

Passing down bindings and restrictions implied by goal conditions can expedite the computation, and turn potentially unsafe programs into safe ones. While top-down binding passing is always a part of SLD-resolution and Prolog, the magic-set and similar methods (Zaniolo et al. 1997) can often produce similar benefits in in the bottom-up execution

model that is typically used for Datalog implementations. To illustrate the extension of the magi-set method to Datalog^{FS} let us consider the following example where we use the goal N =![cbasic(frame)] to find out how many basic parts an assembled frame contains:

Example 1 How many basic components does frame contain?

```
howmanypartsInFrame(N) ← N =![cbasic(frame)].

cassb(Part, Sub):Qty ← assbl(Part, Sub, Qty).
cbasic(Pno):1 ← basic(Pno, _).
cbasic(Part):K ← cassb(Part, Sub), cbasic(Sub),
                 K:[cassb(Part, Sub1), cbasic(Sub1)].
```

Here the binding passing analysis shows that Part in the head of the recursive rule is bound but K is not. Then the body of the recursive rule passes the binding to Sub. Thus the binding passing property hold for the first argument of cbasic. performing the binding passing analysis, our Datalog^{FS} compiler would here produce the following magic-set rules (which only propagate bound values, and thus K is not in the magic-set rules):

```
m.cbasic(frame).
m.cbsasic(Sub) ← m.cassb(Part, Sub), cbasic(Part).
```

and the following modified rules:

```
cbasic(Pno):1 ← basic(Pno, _), m.cbasic(Pno).
cbasic(Part):K ← K:[cassb(Part, Sub), cbasic(Sub),
                    m.cbasic(Part)].
```

Observe that the modified recursive rule must be retained since it is needed to compute the values of K. This example illustrates that the full magic set method must be often applied once the FS test clauses are also considered in the binding passing analysis—whereas, without the FS–test clauses, they could have been compiled as right-linear rules (Zaniolo et al. 1997).

Consider now Example 1 in the paper and say that we ask the question whether a particular individual, denoted by $I will attend, using the goal: ?attend($I). Then, byrecasting its rules into canonical form,

```
attend(X) ← organizer(X).
attend(Y) ← student(Y), attend(X), friend(Y, X),
            3:[attend(X1), friend(Y, X1)].
```

and performing the binding passing analysis, we can apply the magic-set transformation and obtain the following magic set rules

```
m.attend($I).
m.attend(X) ← student(Y), m.attend(Y), friend(Y, X),
              3:[attend(X1), friend(Y, X1)].
```

and the following transformed rules:

```
attend(X) ← organizer(X), m.attend(X).
attend(Y) ← student(Y), attend(X), friend(Y, X),
            3:[attend(X1), friend(Y, X1)], m.attend(Y).
```

The meaning of the magic-set rules here is quite obvious: we chase down the friends of `$I`, and the friends of his friends, and so on and include them in the magic set (an exclude every other person since they are irrelevant to the issue of whether `$I` will attend). But before we include the friends of `X` we also make sure that in fact he/she has at least three friends. The standard differential fixpoint will then be applied to rules so re-written.

A Datalog^{FS} Prototype

A deductive database system supporting all the new monotonic constructs introduced by Datalog^{FS} is the UCLA $\mathcal{DeAL}$ system ($\mathcal{DeAL}$ stands for Deductive Application Language). While much development work remains, the core system is functional and it is scheduled for demonstration at the VLDB 2013 conference (Shkapsky et al. 2013).

The design and development of $\mathcal{DeAL}$ has greatly benefited from the lessons learned from previous Datalog prototypes and in particular from the LDL/LDL++ system, and its deployment in real-life applications written by the less-sophisticated users[1]. Indeed an important challenge faced by $\mathcal{DeAL}$ and its predecessors is that of allowing a wider range of users to take full advantage of the powerful logic-based semantics provided by these declarative languages. To achieve that, we found that the language and its system must present users with concepts and constructs that are intuitive and similar enough to those with which they are familiar with from previous experience. We expect that many of the $\mathcal{DeAL}$ users will come from a data-intensive background and they are thus familiar with SQL and its aggregates, including the COUNT * (UNLIMITED PRECEDING) of SQL 2003, which (i) has an operational semantics that is very similar to our running FS-goals and (ii) is only allowed in the SELECT clause of SQL statements. For this reasons, $\mathcal{DeAL}$ supports the Datalog^{FS}extension via continuous aggregates in the head of the clauses. Therefore, the formal semantics of $\mathcal{DeAL}$ programs are defined by simple syntactic rules that map them back to Datalog^{FS}programs.

Several Datalog languages, proposed in the past, including LDL++, support aggregates in the heads of the program rules. This similarity in syntactic constructs used, and the simple extensions for differential fixpoint and magic-set discussed in this paper, allowed us to base the implementation of $\mathcal{DeAL}$ on the well-tested architecture and efficient implementation techniques of LDL++. For documentation, and information about the system status, availability, and performance on different environments, including parallel ones, the interested reader is referred to [2].

From Integers to Floating-Point Numbers

The max-based optimization allows us to use very large integers for FS-values, without having to repeat this computation for every value up to the max. This property addresses a number of practical issues, including integer arithmetic on very large values causing overflows. While unlimited precision integers can be used to solve this problem, a more efficient approach consists in taking advantage of floating point numbers and their very efficient implementation, although this might result in round-off errors. For instance, if

[1] F. Arni, K. Ong, S. Tsur, H.Wang, and C. Zaniolo. The deductive database system LDL++. In TPLP, 3(1):61–94, 2003

[2] The Deductive Application Language (DeAL) System http://wis.cs.ucla.edu/deals/

the system only supports four-digit precision, then the integer 5222 will be represented as $5.222 \times 10^{+3}$ and the sum of this integer with itself will be rounded off to 1.044×10^4, which is only an approximation for the correct result of 10444. On the other hand, round-offs define monotonic functions on positive integers and they can thus be used in our rules without compromising the least-fixpoint semantics and their amenability to max-based optimization. Thus, by adding to our rules round-off goals that apply to the FS-values, we obtain normal Datalog^{FS} programs where the max-based computation using unlimited-precision integers is approximated by the computation using floating-point numbers. Naturally the user has to decide whether this approximation is satisfactory, or improvements are required, such as using double-precision arithmetic, or using formulas that are less prone to round-off errors. These general computation issues can be addressed by numerical analysis techniques that are outside the scope of this paper.

Supporting Floating-Point Numbers Decimal numbers are required in many applications, that, e.g., deal with percentages and probabilities. All the decimal numbers used in a Datalog^{FS} program can be viewed as the integer numerators of rational numbers that share a given large denominator, D. Then, every arithmetic expression on these rational numbers with common denominator D, can be transformed into equivalent ones that compute the integer numerator of the result represented over the denominator D. For instance, given two numbers N_1/D and N_2/D their sum is $(N_1 + N_2)/D)$. However their product is $((N_1 \times N_2) \div D)/D$, where the integer division can cause a round-off error (e.g., if base-10 numbers are used, and $N_1 = N_2 = 5222$, and $D = 10^6$, then their product is rounded off to $27/10^6$). Modern hardware provides very fast implementations for these computations, whereby the system (i) performs the round-off operation $\div D$ automatically, and (ii) supports very large values for D whereby roundoff errors are minimized. Indeed, we can set $D = 10^C$ where C is the largest value for the negative exponent supported by the system and then the floating point arithmetic emulates exactly the rational number arithmetic described above, where a real number, such as $X = 5.222 \times 10^{-3}$, is simply a compact representation of the rational number $5.222 \times 10^{C-3}/10^C$. For instance, if $C = 12$ the floating point unit computes $X^2 = 2.7269284 \times 10^{-5}$, which corresponds to the rational number $27269284/10^{12}$. Thus, no round-off occurs when $C = 12$. However, when $C = 6$, the floating point unit will produce $2.7 \times 10^{-5} = 27/10^6$ because of the roundoff caused by the underflow. Therefore, the roundoff behavior of floatin-point numbers with least exponent $-C$, produces exactly the same results as those produced by rational number with denominator $D = 10^C$. Current systems support C values where $C \geq 95$ and thus underflow is unlikely to be an issue in most applications. Of course, the emulation that real numbers provide for rational numbers is also limited by the fact that their mantissa is of finite length, and thus for large numbers, we will also incur in the roundoff errors.Therefore, while users must be warned that Datalog^{FS} is not better than any other language in preventing overflow and roundoff problems, positive floating-point numbers can be used freely in normal Datalog^{FS} programs, which have formal semantics, and provide a very efficient implementation through floating-point arithmetic[3]. Using floating-point numbers and real arithmetic, Datalog^{FS} can express a cornucopia of interesting applications.

[3] For simplicity, we have used a decimal base, but the same conclusions hold for other bases.

References

Arni, F., Ong, K., Tsur, S., Wang, H., and Zaniolo, C. 2003. The deductive database system ldl++. *TPLP 3,* 1, 61–94.

Mazuran, M., Serra, E., and Zaniolo, C. 2012. Extending the power of datalog recursion. In *The VLDB Journal*. Springer-Verlag, 1–23.

Seo, J., Guo, S., and Lam, M. S. 2013. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*. 278–289.

Shkapsky, A., Zeng, K., and Zaniolo, C. 2013. Graph queries in a next-generation datalog system. In *VLDB 2013, Demo Track*. 100–104.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S., and Zicari, R. 1997. *Advanced Database Systems*. Morgan Kaufmann.