Evaluation of Logic Programs with Built-Ins and Aggregation: A Calculus for Bag Relations

Matthew Francis-Landau[0000−0002−4139−1111], Tim Vieira[0000−0002−2043−1073], and Jason Eisner[0000−0002−8861−0772]

Johns Hopkins University, Baltimore, MD 21218 USA
{mfl,timv,jason}@cs.jhu.edu

Abstract. We present a scheme for translating logic programs with built-ins and aggregation into algebraic expressions that denote bag relations over ground terms of the Herbrand universe. To evaluate queries against these relations, we develop an operational semantics based on term rewriting of the algebraic expressions. This approach can exploit arithmetic identities and recovers a range of useful strategies, including lazy strategies that defer work until it becomes both possible and necessary. Code is available at https://github.com/matthewfl/dyna-R.

Keywords: Logic Programming, Relational Algebra, Rewriting Systems

1 Introduction

We are interested in developing execution strategies for deductive databases whose defining rules make use of aggregation, recursion, and arithmetic. Languages for specifying such deductive databases are expressive enough that it can be challenging to answer queries against a given database. Term rewriting systems are an attractive approach because they can start with the query-database pair itself as an intensional description of the answer, and then attempt to rearrange it into a more useful extensional description such as a relational table. We will give a notation for algebraically constructing potentially infinite bag relations from constants and built-in relations by unions, joins, projection, aggregation, and recursion. We show how programs in existing declarative languages such as Datalog, pure Prolog, and Dyna can be converted into this notation.

We will then sketch a term rewriting system for simplifying these algebraic expressions, which can in principle be used to answer queries against a bag relation. Term rewriting naturally handles materialization, delayed constraints, constraint propagation, short-circuit evaluation, lazy iteration, and enumerative strategies such as nested-iterator join.

There remains a practical challenge (which is beyond the scope of this paper): determining which rewrites to apply and when, much as in automated theorem proving. Our current implementation is essentially a priority rewrite

system with heuristic priorities and memoization.[1] While an improved execution engine is work in progress, our design includes fair (breadth-first) nondeterministic search, polyvariant specialization through generating new rewrite rules and programmable reduction strategies, static analysis by abstract interpretation, and guess-check-update strategies to solve recursive systems of constraints.

1.1 Approach

Dyna [4] is a generalization of Datalog [1,7] and pure Prolog [3,2]. Our methods apply to all three of these logic programming languages (and more trivially to languages like SQL that can be written in standard relational algebra).

We are given a Herbrand universe $\mathcal{G}$ of ground terms. A Dyna program serves to define a partial map from $\mathcal{G}$ to $\mathcal{G}$, which may be regarded as a set of key-value pairs. Datalog and Prolog are similar, but they can map a key only to `true`, so the program serves only to define the set of keys.

Given a program, a user may query the value of a specific key (ground term). More generally, a user may use a non-ground term to query the values of all keys that match it, so that the answer is itself a set of key-value pairs.

A set of key-value pairs—either a program or the answer to a query—is a relation on two $\mathcal{G}$-valued variables. Our method in this paper will be to describe the desired relation algebraically, building up the description from simpler relations using a relational algebra. These simpler relations can be over any number of variables; they may or may not have functional dependencies as the final key-value relation does; and they may be bag relations, i.e., a given tuple may appear in the relation more than once, or even infinitely many times. Given this description of the desired relation, we will use rewrite rules to simplify it into a form that is more useful to the user. Although we use a **term rewriting system**, we refer to the descriptions being rewritten as **R**-exprs (relational expressions) to avoid confusion with the terms of the object language, Dyna.

1.2 Dyna Examples

To illustrate the task, we first briefly give a couple of examples (adapted from [9]) of useful Dyna programs and queries against them. In §4, we will sketch how to translate Dyna programs into our algebra.

First, we start with a canonical Datalog program written in standard Dyna notation to compute the shortest path in a graph. Additional rules not shown here define the values of `start` and the various `edge` terms.

```
1 path(start) min= 0.
2 path(Sink)  min= path(Source) + edge(Source, Sink).
```

[1] This is already more flexible than Prolog's SLD resolution, which confines its attention at any time to a specific subgoal term and will crash with an "instantiation fault" if that subgoal cannot be rewritten.

This program defines a map with keys such as `edge("albany", "buffalo")`, whose value is the distance from Albany to Buffalo, and keys such as `path("chicago")`, whose value is the total length of the shortest path from `start` to Chicago.

The second rule implies that the value of `path("chicago")` is the minimum value achieved by `path(Source) + edge(Source, "chicago")` for any instantiation of the variable `Source`. If there are no instantiations of `Source` such that `path(Source)` and `edge(Source, "chicago")` both have values, then `path("chicago")` is not a key in the database at all, meaning that Chicago is not reachable at all from `start`.[2]

How is this database used? A user who is located at `start` might query `path("chicago")` to find out how far away it is, or they might query `path(Y)` to find all reachable cities `Y` along with how far away they are. Other queries would return other objects in the database, such as edges.

In the above example, `Sink` and `Source` may range over only a finite set of cities. However, we can easily encounter cases that define infinite relations, as in the following program that runs a simple convolutional neural network:

```
σ(X) = 1/(1+exp(-X)).                          % define sigmoid function
out(J) = σ(in(J)).                              % apply sigmoid function
in(J) += out(I) * edge(I,J).                    % vector-matrix product
in(input(X,Y)) += pixel_brightness(X,Y)                 % external input
loss += (out(J) - target(J))**2.            % L2 loss of the predictions
edge(input(X,Y),hidden(X+DX,Y+DY)) = weight_conv(DX,DY).       % layer 1
edge(hidden(X,Y),output(Z)) = weight_output(X,Y,Z).            % layer 2
weight_output(X,Y,Z) := random(*,-1,1).               % init with random
weight_conv(DX,DY) := random(*,-1,1) for DX:-4..4, DY:-4..4.
```

Without giving a detailed exposition of this program, we point out that it again has `edge` keys, which specify the weighted edges of a neural network. This time, however, the rules that specify such edges define infinitely many of them, in a convolutional structure on an infinite set of neurons.

As a result, a query `edge(I,J)` must return a description of an infinite set of edges. Even so, only finitely many of these edges contribute to the value of `loss`, provided that the input image specifies `pixel_brightness(X,Y)` at only finitely many `(X,Y)` coordinates, and the loss function specifies `target` values for only finitely many neurons. As a result, a clever system will be able to answer a query for `loss` in finite time, essentially by finding the paths between the pixels and the targets (which requires addition/subtraction of `DX` and `DY`) and by determining which of the infinitely many keys `σ(X)` need to compute their values in order to find the `out` activations of the neurons on these paths.

A version of this program indeed runs in our present implementation, although there is not space here to work through such a detailed example. Broadly speaking, the implementation unrolls the definition of `loss` until it is possible to apply rewrites that can make progress on simplifying the definition, for example, by performing arithmetic on known quantities.

[2] Unless `"chicago"` happens to be the value of the `start` key, in which case the first rule comes into play as well. In this case 0 is included in the minimum, so Chicago is always reachable with total distance of at most 0.

2 Syntax and Semantics of **R**-exprs

Let $\mathcal{G}$ be the Herbrand universe of **ground terms** built from a given set $\mathcal{F}$ of ranked functors. We treat constants (including numeric constants) as 0-ary functors. Let $\mathcal{M} = \mathbb{N} \cup \{\infty\}$ be the set of **multiplicities**. A simple definition of a **bag relation** [8] would be a map $\mathcal{G}^n \to \mathcal{M}$ for some n. Such a map would assign a multiplicity to each possible *ordered n-tuple* of ground terms. However, we will use names rather than positions to distinguish the roles of the n objects being related: in our scheme, the n tuple elements will be named by variables.

Let $\mathcal{V}$ be an infinite set of distinguished **variables**. A **named tuple** E is a function mapping some of these variables to ground terms. For any $\mathcal{U} \subseteq \mathcal{V}$, we can write $\mathcal{G}^{\mathcal{U}}$ for the environments $E : \mathcal{U} \mapsto \mathcal{G}$ with domain $\mathcal{U}$. These are just the possible **$\mathcal{U}$-tuples**: that is, tuples over $\mathcal{G}$ whose elements are named by $\mathcal{U}$. This set $\mathcal{G}^{\mathcal{U}}$ of named $\mathcal{U}$-tuples replaces the set $\mathcal{G}^n$ of ordered n-tuples from above.

Below, we will inductively define the set $\mathcal{R}$ of **R**-exprs. The reader may turn ahead to later sections to see some examples. Each **R**-expr `R` has a finite set of **free variables** vars(`R`) $\subseteq \mathcal{V}$, namely the variables that appear in `R` in positions where they are not bound by an operator such as `proj` or `sum`. The idea is for `R` to specify a bag relation over domain $\mathcal{G}$, with columns named by vars(`R`).

The **denotation function** $[\![\cdot]\!]_E$ interprets **R**-exprs in the **environment** E. It defines a multiplicity $[\![\texttt{R}]\!]_E$ for any **R**-expr `R` whose vars(`R`) $\subseteq$ domain(E).

If $\mathcal{U} \supseteq$ vars(`R`), we can dually regard `R` as inducing the map $\mathcal{G}^{\mathcal{U}} \to \mathcal{M}$ defined by $E \mapsto [\![\texttt{R}]\!]_E$. In other words, given $\mathcal{U}$, `R` specifies a bag relation whose column names are $\mathcal{U}$. This is true for *any* $\mathcal{U} \supseteq$ vars(`R`), but the relation constrains only the columns vars(`R`). The other columns can take any values in $\mathcal{G}$. A tuple's multiplicity never depends on its values in those other columns, since our definition of $[\![\cdot]\!]_E$ will ensure that $[\![\texttt{R}]\!]_E$ depends only on the restriction of E to vars(`R`).

We say that `T` is a **term** if $\texttt{T} \in \mathcal{V}$ or $\texttt{T} = f(\texttt{T}_1, \ldots, \texttt{T}_n)$ where $f \in \mathcal{F}$ has rank n and $\texttt{T}_1, \ldots, \texttt{T}_n$ are also terms. Terms typically appear in the object language (e.g., Dyna) as well as in our meta-language (**R**-exprs). Let $\mathcal{T} \supseteq \mathcal{G}$ be the set of terms. Let vars(`T`) be the set of vars appearing in `T`, and extend E in the natural way over terms `T` for which vars(`T`) $\subseteq$ domain(E): $E(f(\texttt{T}_1, \ldots, \texttt{T}_n)) = f(E(\texttt{T}_1), \ldots, E(\texttt{T}_n))$.

We now define $[\![\texttt{R}]\!]_E$ for each type of **R**-expr `R`, thus also presenting the different types of **R**-exprs in $\mathcal{R}$. First, we have **equality constraints** between non-ground terms, which are true in an environment that grounds those terms to be equal. True is represented by multiplicity 1, and false by multiplicity 0.

1. $[\![\texttt{T=U}]\!]_E =$ **if** $E(\texttt{T}) = E(\texttt{U})$ **then** 1 **else** 0, where $\texttt{T}, \texttt{U} \in \mathcal{T}$

We also have **built-in constraints**, such as

2. $[\![\texttt{plus(I,J,K)}]\!]_E =$ **if** $E(\texttt{I}) + E(\texttt{J}) = E(\texttt{K})$ **then** 1 **else** 0, where $\texttt{I}, \texttt{J}, \texttt{K} \in \mathcal{T}$

The above **R**-exprs are said to be **constraints** because they always have multiplicity 1 or 0 in any environment. Taking the **union** of **R**-exprs via `+` may yield larger multiplicities:

3. $[\![\texttt{R+S}]\!]_E = [\![\texttt{R}]\!]_E + [\![\texttt{S}]\!]_E$, where $\texttt{R}, \texttt{S} \in \mathcal{R}$

The **R**-expr `0` denotes the empty bag relation, and more generally, $\texttt{M} \in \mathcal{M}$ denotes the bag relation that contains `M` copies of every $\mathcal{U}$-tuple:

4. $[\![\texttt{M}]\!]_E = \texttt{M}$, where $\texttt{M} \in \mathcal{M}$

To **intersect** two bag relations, we must use multiplication $*$ to combine their multiplicities [8]:

5. $[\![\texttt{R*S}]\!]_E = [\![\texttt{R}]\!]_E \cdot [\![\texttt{S}]\!]_E$, where $\texttt{R}, \texttt{S} \in \mathcal{R}$

Here we regard both `R` and `S` as bag relations over columns $\mathcal{U} \supseteq \text{vars}(\texttt{R}) \cup \text{vars}(\texttt{S}) = \text{vars}(\texttt{R*S}) = \text{vars}(\texttt{R+S})$. The names **intersection**, **join** (or **equijoin**), and **Cartesian product** are conventionally used for the cases of `R*S` where (respectively) $\text{vars}(\texttt{R}) = \text{vars}(\texttt{S})$, $|\text{vars}(\texttt{R}) \cap \text{vars}(\texttt{S})| = 1$, and $|\text{vars}(\texttt{R}) \cap \text{vars}(\texttt{S})| = 0$. As a special case of Cartesian product, notice that `R*3` denotes the same bag relation as `R+R+R`.

We next define **projection**, which removes a named column (variable) from a bag relation, summing the multiplicities of rows (tuples) that have thus become equal. When we translate a logic program into an **R**-expr (§4), we will generally apply projection operators to each rule to eliminate that rule's local variables.

6. $[\![\texttt{proj(X,R)}]\!]_E = \sum_{x \in \mathcal{G}} [\![\texttt{R}]\!]_{E[\texttt{X}=x]}$
   where $\texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$, and where $E[\texttt{X} = x]$ means a version of E that has been modified to put $E(\texttt{X}) = x$ (adding `X` to its domain if it is not already there)

Projection collapses each group of rows that are identical except for their value of `X`. **Summation** does the same, but instead of adding up the multiplicities of the rows in each possibly empty group, it adds up their `X` values to get a `Y` value for the new row, which has multiplicity 1. Thus, it removes column `X` but introduces a new column `Y`. The summation operator is defined as follows (note that $\texttt{sum} \notin \mathcal{F}$):

7. $[\![\texttt{A=sum(X,R)}]\!]_E =$ **if** $E(\texttt{A}) = \sum_{x \in \mathcal{G}} x * [\![\texttt{R}]\!]_{E[\texttt{X}=x]}$ **then** 1 **else** 0
   where $\texttt{A}, \texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$, and the $*$ in the summand means that we sum up $[\![\texttt{R}]\!]_{E[\texttt{X}=x]}$ copies of the value x (perhaps ∞ copies). If there are no summands, the result of the $\sum \cdots$ is defined to be the identity element $\mathsf{id}_{\texttt{sum}}$.

Notice that an **R**-expr of this form is a constraint. `sum` is just one type of **aggregation operator**, based on the binary + operation and its identity element $\mathsf{id}_{\texttt{sum}} = 0$. In exactly the same way, one may define other aggregation operators such as `min`, based on the binary min operation and its identity element $\mathsf{id}_{\texttt{min}} = \infty$. Variants of these will be used in §4 to implement the aggregations in `+=` and `min=` rules like those in §1.2.

In projections `proj(X,R)` and aggregations such as `sum(X,R)` and `min(X,R)`, we say that occurrences of `X` within `R` are **bound** by the projection or aggregation operator,[3] so that they are not in the vars of the resulting **R**-expr. However, the most basic aggregation operator does not need to bind a variable:

8. $[\![\texttt{M=count(R)}]\!]_E =$ **if** $E(\texttt{M}) = [\![\texttt{R}]\!]_E$ **then** 1 **else** 0

[3] But notice that `sum(X,R)` does not correspond to $\sum_{\texttt{X}} \cdots$ but rather to $\sum_{\text{row} \in \texttt{R}} \text{row}[\texttt{X}]$.

In effect, `M=count(R)` is a version of `R` that changes every tuple's multiplicity to 1 but records its original multiplicity in a new column `M`. It is equivalent to `M=sum(N,(N=1)*R)` (where $\mathtt{N} \notin \text{vars}(\mathtt{R})$), but we define it separately here so that it can later serve as an intermediate form in the operational semantics.

Finally, it is convenient to augment the built-in constraint types with **user-defined** relation types. Choose a new functor of rank n that is $\notin \mathcal{F}$, such as `f`, and choose some **R**-expr $\mathtt{R_f}$ with $\text{vars}(\mathtt{R_f}) \subseteq \{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$ (which are n distinct variables) to serve as the **definition** (macro expansion) of `f`. Now define

9. $[\![\mathtt{f}(\mathtt{T}_1, \ldots, \mathtt{T}_n)]\!]_E = [\![\mathtt{R_f}\{\mathtt{X}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{X}_n \mapsto \mathtt{T}_n\}]\!]_E$ where $\mathtt{T}_1, \ldots, \mathtt{T}_n \in \mathcal{T} \cup \mathcal{R}$
   The $\{\mapsto\}$ notation denotes substitution for variables, where bound variables of $\mathtt{R_f}$ are renamed to avoid capturing free variables of the $\mathtt{T}_i$.

With user-defined relation types, it is possible for a user to write **R**-exprs that are circularly defined in terms of themselves or one another (similarly to a `let rec` construction in functional languages). Indeed, a Dyna program normally does this. In this case, the definition of $[\![\cdot]\!]_E$ is no longer a well-founded inductive definition. Nonetheless, we can still interpret the $[\![\cdots]\!]_E = \cdots$ equations in the numbered points above as *constraints* on $[\![\cdot]\!]_E$, and attempt to solve for a denotation function $[\![\cdot]\!]_E$ that satisfies these constraints [4]. Some circularly defined **R**-exprs may be constructed so as to have unique solutions, but this is not the case in general.

3 Rewrite Rules

Where the previous section gave a denotational semantics for **R**-exprs, we now sketch an operational semantics. The basic idea is that we can use rewrite rules to simplify an **R**-expr until it is either a finite materialized relation—a list of tuples—or has some other convenient form. All of our rewrite rules are semantics-preserving, but some may be more helpful than others. For some **R**-exprs that involve infinite bag relations, there may be no way to eliminate all built-in constraints or aggregation operations. The reduced form then includes delayed constraints (just as in constraint logic programming) or delayed aggregators. Even so, conjoining this reduced form with a query can permit further simplification; therefore, some queries may still yield simple answers.

3.1 Finite Materialized Relations

We may express the finite bag relation shown at left by a simple **sum-of-products** **R**-expr, shown at the right. In this example, the ground values being related are integers.

$$g = \left[\begin{array}{cc} \mathtt{X}_1 & \mathtt{X}_2 \\ \hline 1 & 1 \\ 2 & 6 \\ 2 & 7 \\ 2 & 7 \\ 5 & 7 \end{array}\right] \xrightarrow{\text{to } \mathbf{R}\text{-expr}} \begin{array}{rl} \mathtt{R_g} = & (\ (\mathtt{X}_1 = 1) * (\mathtt{X}_2 = 1) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 6) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 7) \\ & + (\mathtt{X}_1 = 2) * (\mathtt{X}_2 = 7) \\ & + (\mathtt{X}_1 = 5) * (\mathtt{X}_2 = 7)\ ) \end{array} \tag{1}$$

We see that each individual row of the table (tuple) translates into a product ($*$) of several (`Variable = value`) expressions, where the variable is the column's name and the value is the cell in the table. To encode multiple rows, the **R**-expr simply adds the **R**-exprs for the individual rows. When evaluated in a given environment, the **R**-expr is a sum of products of multiplicities. But abstracting over possible environments, it represents a union of Cartesian products of 1-tuples, yielding a bag relation.

We may use this **R**-expr as the basis of a new user-defined **R**-expr type `g` (case 9 of §2) by taking its definition $\texttt{R}_\texttt{g}$ to be this **R**-expr. Our **R**-exprs can now include constraints such as `g(J,K)` or `g(J,7)`. When adding a new case in the denotational semantics in this way, we always match it in the operational semantics by introducing a corresponding rewrite rule $\texttt{g}(\texttt{X}_1, \texttt{X}_2) \to \texttt{R}_\texttt{g}$.

A sum-of-products **R**-expr simply enumerates the tuples of a bag relation, precisely as a boolean expression in disjunctive normal form (that is, an "or-of-ands" expression) enumerates the satisfying assignments. Just as in the boolean case, a disjunct does not have to constrain every variable: it may have "don't care" elements. For example, `(J=1)*(K=1) + (J=2)` describes an infinite relation because the second disjunct `J=2` is true for any value of `K`.

3.2 Equality Constraints and Multiplicity Arithmetic

We may wish to query whether the relation g in the above section relates 2 to 7. Indeed it does—and twice. We may discover this by considering `g(2,7)`, which rewrites immediately via substitution to an **R**-expr that has no variables at all (`(2=1)*(7=1)+` $\cdots$), and finding that this further reduces to the multiplicity `2`.

How does this reduction work? First, we need to know how to evaluate the equality constraints: we need to rewrite `2=1` $\to$ `0` but `2=2` $\to$ `1`. The necessary rewrite rules are special cases of the following structural unification rules:

$\texttt{(}f\texttt{(}\texttt{U}_1\texttt{,}\ldots\texttt{,}\texttt{U}_n\texttt{)=}g\texttt{(}\texttt{V}_1\texttt{,}\ldots\texttt{,}\texttt{V}_m\texttt{))} \to \texttt{0}$ **if** $f, g \in \mathcal{F}$ **and** $(f,n) \neq (g,m)$ ▷*functor clash*

$\texttt{(}f\texttt{(}\texttt{U}_1\texttt{,}\ldots\texttt{,}\texttt{V}_n\texttt{)=}f\texttt{(}\texttt{V}_1\texttt{,}\ldots\texttt{,}\texttt{V}_n\texttt{))} \to \texttt{(}\texttt{U}_1\texttt{=}\texttt{V}_1\texttt{)} * \cdots * \texttt{(}\texttt{U}_n\texttt{=}\texttt{V}_n\texttt{)}$ **if** $f \in \mathcal{F}$ **and** $n \geq 0$

`(T=X)` $\to$ `(X=T)` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ ▷*put var on left to match rules below*

`(X=X)` $\to$ `1` **if** `X` $\in \mathcal{V}$ ▷*because true for every value of* `X`

`(X=T)` $\to$ `0` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ **and** `X` $\in \text{vars}($`T`$)$ ▷*not true for any* `X` *(occurs check)*

We now have an arithmetic expression, which we can simplify to the multiplicity `2` via rewrites that implement basic arithmetic on multiplicities $\mathcal{M}$:

`M + N` $\to$ `L` **if** `M,N` $\in \mathcal{M}$ **and** `M+N=L`; `M * N` $\to$ `L` **if** `M,N` $\in \mathcal{M}$ **and** `M*N=L`

Above, we relied on the definition of the new relation type `g`, which allowed us to request a specialization of $\texttt{R}_\texttt{g}$. Do we need to make such a definition in order to query a given bag relation? No: we may do so by conjoining the **R**-expr with additional constraints. For example, to get the multiplicity of the pair $(2,7)$ in $\texttt{R}_\texttt{g}$, we may write $\texttt{R}_\texttt{g}\texttt{*(}\texttt{X}_1\texttt{=2)*(}\texttt{X}_2\texttt{=7)}$. This filters the original relation to just the pairs that match $(2,7)$, and simplifies to $\texttt{2*(}\texttt{X}_1\texttt{=2)*(}\texttt{X}_2\texttt{=7)}$. To accomplish this simplification, we need to use the following crucial rewrite:

`(X=T)*R` $\to$ `(X=T)*R{X` $\mapsto$ `T}` **if** `X` $\in \mathcal{V}$ **and** `T` $\in \mathcal{T}$ ▷*equality propagation*

As a more interesting example, the reader may consider simplifying the query $\texttt{R}_\texttt{g}\texttt{*lessthan(}\texttt{X}_2\texttt{,7)}$, which uses a built-in inequality constraint (see §3.4).

3.3 Joining Relations

Analogous to eq. (1), we define a second tabular relation f with a rewrite rule $\texttt{f}(\texttt{X}_1, \texttt{X}_2) \to \texttt{R}_\texttt{f}$.

$$f = \left[\begin{array}{cc} \texttt{X}_1 & \texttt{X}_2 \\ \hline 1 & 2 \\ 3 & 4 \end{array}\right] \xrightarrow{\text{to } \mathbf{R}\text{-expr}} \texttt{R}_\texttt{f} = \begin{array}{c} (\texttt{X}_1 = 1) * (\texttt{X}_2 = 2) \\ + (\texttt{X}_1 = 3) * (\texttt{X}_2 = 4) \end{array} \tag{2}$$

We can now consider simplifying the **R**-expr `f(I,J) * g(J,K)`, which the reader may recognize as an equijoin on f's second column and g's first column.[4] Notice that the **R**-expr has *renamed* these columns to its own free variables (`I`, `J`, `K`). Reusing the variable `J` in each factor is what gives rise to the join on the relevant column. (Compare `f(I,J) * g(J',K)`, which does not share the variable and so gives the Cartesian product of the f and g relations.)

We can "materialize" the equijoin by reducing it to a sum-of-products form as before, if we wish: `(I=1)*(J=2)*(K=6) + 2*(I=1)*(J=2)*(K=7)`.

To carry out such simplifications, we use the fact that multiplicities form a commutative semiring under `+` and `*`. Since any **R**-expr evaluates to a multiplicity, these rewrites can be used to rearrange unions and joins of **R**-exprs $\texttt{Q}, \texttt{R}, \texttt{S} \in \mathcal{R}$:

`1 * R` $\leftrightarrow$ `R`	▷*multiplicative identity*
`0 * R` $\leftrightarrow$ `0`	▷*multiplicative annihilation*
`0 + R` $\leftrightarrow$ `R`	▷*additive identity*
∞ `* R` $\to \infty$ **if** `R` $\in \mathcal{M}$ **and** `R` $>$ `0`	▷*absorbing element*
∞ `+ R` $\to \infty$	▷*absorbing element*
`R + S` $\leftrightarrow$ `S + R`; `R * S` $\leftrightarrow$ `S * R`	▷*commutativity*
`Q + (R + S)` $\leftrightarrow$ `(Q + R) + S`; `Q * (R * S)` $\leftrightarrow$ `(Q * R) * S`	▷*associativity*
`Q * (R + S)` $\leftrightarrow$ `Q * R + Q * S`	▷*distributivity*
`R * M` $\to$ `R + (R * N)` **if** `M,N` $\in \mathcal{M}$ **and** `(1+N` $\to$ `M)`	▷*implicitly does* `M` $\to$ `1+N`
`R` $\leftrightarrow$ `R * R` **if** `R` is a constraint	▷*as defined in §2*

We can apply some of these rules to simplify our example as follows:

```
f(I,J)*g(J,K)
→ ((I=1)*(J=2)+(I=3)*(J=4)) * g(J,K)                    ▷eq. (2)
→ (I=1)*(J=2)*g(J,K) + (I=3)*(J=4)*g(J,K)               ▷distributivity
→ (I=1)*(J=2)*g(2,K) + (I=3)*(J=4)*g(4,K)               ▷equality propagation
→ (I=1)*(J=2)*( (K=6)+(K=7)*2 ) + (I=3)*(J=4)*0         ▷via eq. (1)
→ (I=1)*(J=2)*( (K=6)+(K=7)*2 )                         ▷annihilation
→ (I=1)*(J=2)*(K=6) + (I=1)*(J=2)*(K=7)*2               ▷distributivity
```

Notice that the factored intermediate form `(I=1)*(J=2)*( (K=6)*1 + (K=7)*2 )` is more compact than the final sum of products, and may be preferable in some settings. In fact, it is an example of a **trie** representation of a bag relation. Like the root node of a trie, the expression partitions the bag of (`I`,`J`,`K`) tuples into disjuncts according to the value of `I`. Each possible value of `I` (in this case only `I=1`) is multiplied by a trie representation of the bag of (`J`,`K`) tuples that can co-occur

[4] This notation may be familiar from Datalog, except that we are writing the conjunction operation as `*` rather than with a comma, to emphasize the fact that we are multiplying multiplicities rather than merely conjoining booleans.

with this `I`. That representation is a sum over possible values of `J` (in this case only `J=2`), which recurses again to a sum over possible values of `K` (`K=6` and `K=7`). Finally, the multiplicities `1` and `2` are found at the leaves. A trie-shaped **R**-expr generally has a smaller branching factor than a sum-of-products **R**-expr. As a result, it is comparatively fast to query it for all tuples that strongly restrict `I` or `(I,J)` or `(I,J,K)`, by narrowing down to the matching summand(s) at each node. For example, multiplying our example trie by the query `I=5` gives an **R**-expr that can be immediately simplified to `0`, as the single disjunct (for `I=1`) does not match.

That example query also provides an occasion for a larger point. This trie simplification has the form `(I=5)*(I=1)*R`, an expression that in general may be simplified to `0` on the basis of the first two factors, without spending any work on simplifying the possibly large expression `R`. This is an example of **short-circuiting** evaluation—the same logic that allows a SAT solver or Prolog solver to backtrack immediately upon detecting a contradiction.

3.4 Rewrite Rules for Built-In Constraints

Built-in constraints are an important ingredient in constructing infinite relations. While they are not the only method,[5] they have the advantage that libraries of built-in constraints such as `plus(I,J,K)` (case 2 of §2) usually come with rewrite rules for reasoning about these constraints [6]. Some of the rewrite rules invoke opaque procedural code.

Recall that the arguments to a `plus` constraint are terms, typically either variables or numeric constants. Not all `plus` constraints can be rewritten, but a library should provide at least the following cases:

$$\texttt{plus(I,J,K)} \to \underbrace{\mathbb{I}(\texttt{I} + \texttt{J} = \texttt{K})}_{\in\{0,1\}} \textbf{ if } \texttt{I},\texttt{J},\texttt{K} \in \mathbb{R}$$

$$\texttt{plus(I,J,X)} \to \texttt{(X=I + J)} \textbf{ if } \texttt{I},\texttt{J} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(I,X,K)} \to \texttt{(X=K - I)} \textbf{ if } \texttt{I},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(X,J,K)} \to \texttt{(X=}\underbrace{\texttt{K - J}}_{\in\mathbb{R}}\texttt{)} \textbf{ if } \texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

The **R**-expr `R = proj(J,plus(I,3,J)*plus(J,4,K))` represents the infinite set of `(I,K)` pairs such that `K = (I+3)+4` arithmetically. (The intermediate temporary variable `J` is projected out.) The rewrite rules already presented (plus a rewrite rule from §3.5 below to eliminate `proj`) suffice to obtain a satisfactory answer to the query `I=2` or `K=9`, by reducing either `(I=2)*R` or `R*(K=9)` to `(I=2)*(K=9)`.

On the other hand, if we wish to reduce `R` itself, the above rules do not apply. In the jargon, the two `plus` constraints within `R` remain as **delayed constraints**, which cannot do any work until more of their variable arguments are replaced by constants (e.g., due to equality propagation from a query, as above).

We can do better in this case with a library of additional rewrite rules that implement standard axioms of arithmetic [6], in particular the associative law. With these, `R` reduces to `plus(I,7,K)`, which is a simpler description of this infinite relation. Such rewrite rules are known as **constraint propagators**. Other

[5] Others are structural equality constraints and recursive user-defined constraints.

useful examples concerning `plus` include `plus(0,J,K)` → `K=J` and `plus(I,J,J)` → `(I=0)`, since unlike the rules at the start of this section, they can make progress even on a single `plus` constraint whose arguments include more than one variable. Similarly, some useful constraint propagators for the `lessthan` relation include `lessthan(J,J)` → `0`; the transitivity rule `lessthan(I,J)*lessthan(J,K)` → `lessthan(I,J)*lessthan(J,K)*lessthan(I,K)`; and `lessthan(0,I)*plus(I,J,K)` → `lessthan(0,I)*plus(I,J,K)*lessthan(J,K)`. The integer domain can be **split** by rules such as `int(I)` → `int(I)*(lessthan(0,I)+lessthan(I,1))` in order to allow case analysis of, for example, `int(I)*myconstraint(I)`. All of these rules apply even if their arguments are variables, so they can apply early in a reduction before other rewrites have determined the values of those variables. Indeed, they can sometimes short-circuit the work of determining those values.

Like all rewrites, built-in rewrites `R` → `S` must not change the denotation of `R`: they ensure $[\![R]\!]_E = [\![S]\!]_E$ for all E. For example, `lessthan(X,Y)*lessthan(Y,X)` →* `0` is semantics-preserving because both forms denote the empty bag relation.

3.5 Projection

Projection is implemented using the following rewrite rules. The first two rules make it possible to push the `proj(X,...)` operator down through the sums and products of `R`, so that it applies to smaller subexpressions that mention `X`:

`proj(X,R+S)` ↔ `proj(X,R) + proj(X,S)` ▷*distributivity over +*
`proj(X,R*S)` ↔ `R*proj(X,S)` **if** `X` ∉ vars(`R`) ▷*see also the* `R*`∞ *rule below*

Using the following rewrite rules, we can then eliminate the projection operator from smaller expressions whose projection is easy to compute. (In other cases, it must remain as a delayed operator.) How are these rules justified? Observe that `proj(X,R)` in an environment E denotes the number of `X` values that are consistent with E's binding of `R`'s other free variables. Thus, we may safely rewrite it as another expression that always achieves the same denotation.

`proj(X,(X=T))` → 1 **if** `T` ∈ $\mathcal{T}$ **and** `X` ∉ vars(`T`) ▷*occurs check*
`proj(A,(A=sum(X,R)))` → 1 **if** `A` ∉ vars(`R`) ▷*cardinality of an aggregated variable*
`proj(X,R)` → `R*`∞ **if** `X` ∉ vars(`R`) ▷*cardinality of an unconstrained variable*
`proj(X,bool(X))` → 2 ▷*cardinality of a variable given a certain unary constraint*
`proj(X,int(X))` → ∞ ▷*cardinality of a variable given a certain unary constraint*
`proj(X,proj(Y,nand(X,Y)))` → 3 ▷*card. of a pair given a certain binary constraint*

As a simple example, let us project column `K` out of the table `g(J,K)` from eq. (1).

```
                    proj(K,((J=1) * (K=1)        ( (J=1) * proj(K, (K=1))
                          + (J=2) * (K=6)        + (J=2) * proj(K, (K=6))
 proj(K, g(J,K)) →        + (J=2) * (K=7)   →    + (J=2) * proj(K, (K=7))
                          + (J=2) * (K=7)        + (J=2) * proj(K, (K=7))
                          + (J=5) * (K=7)  ))    + (J=5) * proj(K, (K=7)) )
→ (J=1) + (J=2)*3 + (J=5)
```

When multiple projection operators are used, we may push them down independently of each other, since they commute:

`proj(X, proj(Y, R))` → `proj(Y, proj(X, R))`

3.6 Aggregation

The simple `count` aggregator from §2 is implemented with the following rewrite rules, which resemble those for `proj`:

```
M=count(R+S) → proj(L,(L=count(R)) * proj(N,(N=count(S)) * plus(L,N,M)))
M=count(R*S) → proj(L,(L=count(R)) * proj(N,(N=count(S)) * times(L,N,M))) if
      vars(R) ∩ vars(S) = ∅
M=count(N) → (M=N) if N ∈ 𝓜
```

In the first two rules, `L` and `N` are new bound variables introduced by the right-hand side of the rule. (When the rewrite rule is applied, they will—as is standard—potentially be renamed to avoid capturing free variables in the arguments to the left-hand side.) They serve as temporary registers. The third rule covers the base case where the expression has been reduced to a constant multiplicity: e.g.,

```
M=count(5=5) → M=count(1) → M=1
M=count(plus(I,J,J)*(I=5)) → M=count((I=0)*(I=5)) →* M=count(0) → M=0
```

The following rewrite rules implement `sum`. (The rules for other aggregation operators are isomorphic.) The usual strategy is to rewrite `A=sum(X,R)` as a chain of `plus` constraints that maintain a running total. The following rules handle cases where `R` is expressed as a union of 0, 1, or 2 bag relations, respectively. (A larger union can be handled as a union of 2 relations, e.g., `(Q+R)+S`.)

```
A=sum(X, 0) → (A=id_sum)
A=sum(X, (X=T)) → (A=T) if T ∈ 𝓣 and X ∉ vars(T)                ▷occurs check
A=sum(X, R+S) → proj(B, (B=sum(X,R)) * proj(C, (C=sum(X,S)) * plus(B,C,A)))
```

The second rule above handles only one of the base cases of 1 bag relation. We must add rules to cover other base cases, such as these:[6]

```
A=sum(X, (X=sum(Y,R))) → (A=sum(Y,R)) if X ∉ vars(R)
A=sum(X, (X=min(Y,R))) → (A=min(Y,R)) if X ∉ vars(R)
```

Most important of all is this case, which is analogous to the second rule of §3.5 and is needed to aggregate over sum-of-products constructions:

```
A=sum(X,R*S) ↔ sum_copies(R,B,A)*(B=sum(X,S))   if X ∉ vars(R)
```

Here, `sum_copies(M,B,A)` for `M` $\in \mathcal{M}$ constrains `A` to be the aggregation of `M` $\in \mathcal{M}$ copies of the aggregated value `B`. The challenge is that in the general case we actually have `sum_copies(R,B,A)`, so the multiplicity `M` may vary with the free variables of `R`. The desired denotational semantics are

$$[\![\texttt{sum\_copies(R,B,A)}]\!]_E = \textbf{ if } [\![\texttt{R}]\!]_E \cdot [\![\texttt{B}]\!]_E = [\![\texttt{A}]\!]_E \textbf{ then } 1 \textbf{ else } 0$$
$$[\![\texttt{min\_copies(R,B,A)}]\!]_E = \textbf{ if } ([\![\texttt{R}]\!]_E = 0 \textbf{ and } [\![\texttt{A}]\!]_E = \mathsf{id}_{\texttt{min}}) \textbf{ or } ([\![\texttt{R}]\!]_E > 0 \textbf{ and } [\![\texttt{A}]\!]_E = [\![\texttt{B}]\!]_E) \textbf{ then } 1 \textbf{ else } 0$$

where `R` $\in \mathcal{R}$ and `B`, `A` $\in \mathcal{T}$

[6] As in §3.5, we could also include special rewrites for certain aggregations that have a known closed-form result, such as certain series sums.

where we also show the interesting case of `min_copies(M,B,A)`, which is needed to help define the `min` aggregator. We can implement these by the rewrite rules

`sum_copies(R,B,A)` $\rightarrow$ `proj(M,(M=count(R))*times(M,B,A))` $\triangleright$ *assumes* $\mathsf{id}_{\mathtt{sum}} = 0$
`min_copies(R,B,A)` $\rightarrow$ `proj(M,(M=count(R))*((M=0)*(A=`$\mathsf{id}_{\mathtt{min}}$`)+lessthan(0,M)*(A=B)))`

Identities concerning aggregation yield additional rewrite rules. For example, since multiplication distributes over $\sum$, summations can be merged and factored via `(B=sum(I,R))*(C=sum(J,S))*times(B,C,A)` $\leftrightarrow$ `A=sum(K,R*S*times(I,J,K))` provided that `I` $\in$ vars(`R`), `J` $\in$ vars(`S`), `K` $\notin$ vars(`R*S`), and vars(`R`) $\cap$ vars(`S`) $= \emptyset$. Other distributive properties yield more rules of this sort. Moreover, projection and aggregation operators commute if they are over different free variables.

To conclude this section, we now attempt aggregation over infinite streams. We wish to evaluate `A=exists(B,proj(I,peano(I)*myconstraint(I)*(B=true)))` to determine whether there exists any Peano numeral that satisfies a given constraint. Here `exists` is the aggregation operator based on the binary `or` operation.

`peano(I)` represents the infinite bag of Peano numerals, once we define a user constraint via the rewrite rule `peano(I)` $\rightarrow$ `(I=zero) + proj(J,(I=s(J))*peano(J))`. Rewriting `peano(I)` provides an opportunity to apply the rule again (to `peano(J)`). After $k \geq 0$ rewrites we obtain a representation of the original bag that explicitly lists the first k Peano numerals as well as a continuation that represents all Peano numerals $\geq k$:

$$\rightarrow \texttt{(I=zero) + } \cdots \texttt{ + (I=}\underbrace{\texttt{s(}\cdots\texttt{s(}}_{k-1 \text{ times}}\texttt{zero)}\cdots\texttt{))} \texttt{ + } \overbrace{\texttt{proj(J,(I=}\underbrace{\texttt{s(}\cdots\texttt{s(}}_{k-1 \text{ times}}\texttt{J)}\cdots\texttt{)))*peano(J)}}^{\text{continuation}}$$

Rewriting the `exists` query over this $(k+1)$-way union results in a chain of k `or` constraints. If one of the first k Peano numerals in fact satisfies `myconstraint`, then we can "short-circuit" the infinite regress and determine our answer without further expanding the continuation, thanks to the useful rewrite `or(true,C,A)` $\rightarrow$ `(A=true)`, which can apply even while `C` remains unknown.

In general, one can efficiently aggregate a function of the Peano numerals by alternating between expanding `peano` to draw the next numeral from the iterator, and rewriting the aggregating **R**-expr to aggregate the value of the function at that numeral into a running "total." If the running total ever reaches an absorbing element a of the aggregator's binary operation—such as `true` for the `or` operation—then one will be able to simplify the expression to `A=` a and terminate. We leave the details as an exercise.

4 Translation of Dynabases to R-exprs

The translation of a Dyna program to a single recursive **R**-expr can be performed mechanically. We will illustrate the basic construction on the small contrived example below. We will focus on the first three rules, which define `f` in terms of `g`. The final rule, which defines `g`, will allow us to take note of a few subtleties.

```
12 f(X) += X*X.
13 f(4) += 3.
14 f(X) += g(X,Y).
15 g(4*C,Y) += C-1 for Y > 99.
```

Recall that a Dyna program represents a set of key-value pairs. `is(Key,Val)` is the conventional name for the key-value relation. The above program translates into the following user-defined constraint, which recursively invokes itself when the value of one key is defined in terms of the values of other keys.

```
is(Key,Val) → (Val=sum(Result,                ▷sum represents the += aggregator
    proj(X, (Key=f(X))*times(X,X,Result) )                    ▷f(X) += X*X.
  +         (Key=f(4))*(Result=3)                               ▷f(4) += 3.
  +proj(X, (Key=f(X))*proj(Y,is(g(X,Y),Result))) )          ▷f(X) += g(X,Y).
  +proj(C, proj(Y, proj(Temp,(Key=g(Temp,Y))*times(4,C,Temp))   ▷g(4*C,Y) +=
                *minus(C,1,Result)*lessthan(99,Y)     ▷... C-1 for Y > 99.
 ) * notnull(Val)         ▷notnull discards any pair whose Val aggregated nothing
```

Each of the 4 Dyna rules translates into an **R**-expr (as indicated by the comments above) that describes a bag of (`Key`,`Result`) pairs of ground terms. In each pair, `Result` represents a possible ground value of the rule's body (the Dyna expression to the right of the aggregator `+=`), and `Key` represents the corresponding grounding of the rule head (the term to the left of the aggregator), which will receive `Result` as an aggregand. Note that the same (`Key`,`Result`) pair may appear multiple times. Within each rule's **R**-expr, we project out the variables such as `X` and `Y` that appear locally within the rule, so that the **R**-expr's free variables are only `Key` and `Result`.[7]

Dyna mandates in-place evaluation of Dyna expressions that have values [4]. For each such expression, we create a new local variable to bind to the result. Above, the expressions such as `X*X`, `g(X,Y)`, `4*C`, and `C-1` were evaluated using `times`, `is`, `times`, and `minus` constraints, respectively, and their results were assigned to new variables `Result`, `Result`, `Temp`, and `Result`. Importantly, `g(X,Y)` refers to a key of the Dyna program itself, so the Dyna program translated into an `is` constraint whose definition recursively invokes `is(g(X,Y),Result)`.

The next step is to take the bag union (`+`) of these 4 bag relations. This yields the bag of all (`Key`,`Result`) pairs in the program. Finally, we wrap this combined bag in `Val=sum(Result,...)` to aggregate the `Result`s for each `Key` into the `Val` for that key. This yields a set relation with exactly one value for each key. For `Key=f(4)`, for example, the first and second rules will each contribute one `Result`, while the third rule will contribute as many `Result`s as the map has keys of the form `g(4,Y)`.[8]

We use `sum` as our aggregation operator because all rules in this program specify the `+=` aggregator. One might expect `sum` to be based on the binary operator that is implemented by the `plus` builtin, as described before, with identity element $\mathsf{id}_{\texttt{sum}} = 0$. There is a complication, however: in Dyna, a `Key` that has

[7] It is always legal to project out the local variables at the top level of the rule, e.g., `proj(X,proj(Y,...))` for rule 3. However, we have already seen rewrite rules that can narrow the scope of `proj(Y,...)` to a sub-**R**-expr that contains all the copies of `Y`. Here we display the version after these rewrites have been done.

[8] The reader should be able to see that the third Dyna rule will contribute infinitely many copies of `0`, one for each `Y > 99`. This is an example of multiplicity ∞. Fortunately, `sum_copies` (invoked when rewriting the `sum` aggregation operator) knows that summing any positive number of `0` terms—even infinitely many—will give `0`.

no `Result`s should not in fact be paired with `Val=0`. Rather, this `Key` should not appear as a key of the final relation at all! To achieve this, we augment $\mathcal{F}$ with a new constant `null` (similar to Prolog's `no`), which represents "no results." We define $\mathsf{id}_{\texttt{sum}} = \mathsf{null}$, and we base `sum` on a modified version of `plus` for which `null` is indeed the identity (so the constraints `plus(null,X,X)` and `sum_copies(0,X,null)` are both true for all `X`). All aggregation operators in Dyna make use of `null` in this way. Our `Val=sum(Result,...)` relation now obtains `null` (rather than `0`) as the unique `Val` for each `Key` that has no aggregands. As a final step in expressing a Dyna program, we always remove from the bag all (`Key`, `Val`) pairs for which `Val=null`, by conjoining with a `notnull(Val)` constraint. This is how we finally obtain the **R**-expr above for `is(Key,Val)`.

To query the Dyna program for the value of key `f(4)`, we can reduce the expression `is(f(4),Val)`, using previously discussed rewrite rules. We can get as far as this before it must carry out its own query `g(4,Y)`:

```
proj(C, (C=sum(Result,proj(Y,is(g(4,Y),Result))))
       * plus(19,C,Val)                          ) * notnull(Val)
```

where the local variable `C` captures the total contribution from rule 3, and `19` is the total contribution of the other rules. To reduce further, we now recursively expand `is(g(4,Y),Result)` and ultimately reduce it to `Result=0` (meaning that `g(4,Y)` turns out to have value `0` for all ground terms `Y`). `proj(Y,Result=0)` reduces to `(Result=0)*∞`—a bag relation with an infinite multiplicity—but then `C=sum(Result,(Result=0)*∞)` reduces to `C=0` (via `sum_copies`, as footnote 8 noted). The full expression now easily reduces to `Val=19`, the correct answer.

What if the Dyna program has different rules with different aggregators? Then our translation takes the form

```
is(Key,Val) → Val=only(Val1,  (Val1=sum(Result, RSum))
                            + (Val1=min(Result, RMin))
                            + (Val1=only(Result, REq))
                            + ...                     ) * notnull(Val)
```

where `RSum` is the bag union of the translated `+=` rules as in the previous example, `RMin` is the bag union of the translated `min=` rules, `REq` is the bag union of the translated `=` rules, and so on. The new aggregation operator `only` is based on a binary operator that has identity $\mathsf{id}_{\texttt{only}} = \mathsf{null}$ and combines any pair of non-`null` values into `error`. For each `Key`, therefore, `Val` is bound to the aggregated result `Val1` of the *unique* aggregator whose rules contribute results to that key. If multiple aggregators contribute to the same key, the value is `error`.[9]

A Dyna program may consist of multiple *dynabases* [4]. Each dynabase defines its own key-value mapping, using aggregation over only the rules in that dynabase, which may refer to the values of keys in other dynabases. In this case, instead of defining a single constraint `is` as an **R**-expr, we define a different named constraint for each dynabase as a different **R**-expr, where each of the **R**-exprs may call any of these named constraints.

[9] A Dyna program is also supposed to give an `error` value to the key `a` if the program contains both the rules `a = 3` and `a = 4`, or the rule `a = f(X)` when both `f(0)` and `f(1)` have values. So we also used `only` above as the aggregation operator for `=` rules.

5 Conclusions and Ongoing Work

We have shown how to algebraically represent the bag-relational semantics of any program written in a declarative logic-based language like Dyna. A query against a program can be evaluated by joining the query to the program and simplifying the resulting algebraic expression with appropriate term rewriting rules. It is congenial that this approach allows evaluation to flexibly make progress, propagate information through the expression, exploit arithmetic identities, and remove irrelevant subexpressions rather than wasting possibly infinite work on them.

In ongoing work, we are considering methods for practical interpretation and compilation of rewrite systems. Our goal is to construct an evaluator that will both perform static analysis and "learn to run fast" [9] by constructing or discovering reusable strategies for reducing expressions of frequently encountered sorts. In the case of cyclic rewrite systems, the system should be able to guess portions of a solution and then verify that the guesses are consistent with the rewrite rules. A forward chaining mechanism can be used to invalidate or correct inconsistent guesses, as is common in Datalog, and this mechanism can also be used for change propagation when the program or the query is externally updated [5].

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant No. 1629564. We thank Scott Smith for helpful comments, and the WRLA program chairs Santiago Escobar and Narciso Martí Oliet for allowing generous time to revise the paper.

References

1. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). In: IEEE Transactions on Knowledge and Data Engineering (1989)
2. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer-Verlag (1984)
3. Colmerauer, A., Roussel, P.: The Birth of Prolog, chap. 7. Association for Computing Machinery (1996)
4. Eisner, J., Filardo, N.W.: Dyna: Extending Datalog for modern AI. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded. Lecture Notes in Computer Science, Springer (2011)
5. Filardo, N.W., Eisner, J.: A flexible solver for finite arithmetic circuits. In: Technical Communications of the International Conference on Logic Programming. Leibniz International Proceedings in Informatics (LIPIcs) (2012)
6. Frühwirth, T.: Theory and practice of constraint handling rules. The Journal of Logic Programming **37**(1) (1998)
7. Gallaire, H., Minker, J., Nicolas, J.M.: Logic and databases: A deductive approach. ACM Comput. Surv. **16**(2) (1984)
8. Green, T.J.: Bag semantics. In: LIU, L., ÖZSU, M.T. (eds.) Encyclopedia of Database Systems. Springer (2009)
9. Vieira, T., Francis-Landau, M., Filardo, N.W., Khorasani, F., Eisner, J.: Dyna: Toward a self-optimizing declarative language for machine learning applications. In: Proceedings of the ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL). ACM (2017)