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

Matthew Francis-Landau and Tim Vieira and Jason Eisner

Johns Hopkins University, Baltimore Maryland, USA
(*e-mail:* {mfl,timv,jason}@cs.jhu.edu)

Abstract

We present a scheme for translating logic programs, which may use aggregation and arithmetic, 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 possible or necessary.[1]

KEYWORDS: Logic Programming, Relational Algebra, Term Rewriting

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 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. Our current implementation[1] is an interpreter that manages the application of these rewrites using a specific strategy outlined in §B (along with a memoization facility that is beyond the scope of this paper). We are also exploring the use of machine learning to explore rewriting strategies (Vieira et al., 2017).

1.1 Approach

Dyna (Eisner and Filardo, 2011) is a generalization of Datalog (Ceri et al., 1989; Gallaire et al., 1984) and pure Prolog (Colmerauer and Roussel, 1996; Clocksin and Mellish, 1984). Our methods

[1] This manuscript is an expanded version of a paper that was presented at the 13th International Workshop on Rewriting Logic and Its Applications (WRLA 2020). Code is available at https://github.com/matthewfl/dyna-R.

apply to all three of these logic programming languages. Our methods could also be used on simpler languages like SQL that correspond to standard relational algebra, which we generalize.

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. A Datalog or Prolog program is similar, but it can map a key in $\mathcal{G}$ only to `true`, so it 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 The Dyna Language

Datalog is a logical notation for defining finite database relations. Using Prolog's pattern-matching notation, a Datalog program can concisely define each relation from other relations. The notation makes it easy to express constants, don't-cares, union, selection, and join. For example, a join of relations `b` and `c` is defined by the Datalog rule `a(I,K) :- b(I,J), c(J,K)`, where the capitalized identifiers are variables. The `:-` operator is read as "if," and the top-level comma is read as "and." Thus, if `b(2,8)` and `c(8,5)` have already been established to be true, then this rule will establish that the ground term `a(2,5)` $\in \mathcal{G}$ is true—that is, the defined relation `a` contains the tuple $(2,5)$.

While Datalog permits relations to be defined recursively and sometimes provides access to particular infinite arithmetic relations, certain restrictions in Datalog ensure that all new relations defined by a user program remain finite and hence materializable. Dyna drops these restrictions to allow the definition of infinite relations as in Prolog.

Dyna also provides an attractive notation for aggregation. A Dyna rule is not a **Horn clause** (as above) but a **Horn equation** such as `a(I,K) += b(I,J) * c(J,K)`. This rule can be read as implementing a generalized form of the matrix multiplication $(\forall i,j) a_{ik} = \sum_j b_{ij} \cdot c_{jk}$. It is "generalized" because `I`, `J`, `K` range not over some finite set of integers, but over the entire Herbrand universe $\mathcal{G}$. Their values may be any constant terms, including structured terms such as lists, perhaps representing real-world entities. Nonetheless, for a given `I` and `K`, it may be that `a(I,K)` has only finitely many summands, because summands without values are dropped, as we will see in a moment.

The difference from Horn clauses is that `a(I,K) += b(I,J) * c(J,K)` is used not to establish that `a(2,5)` is true, but that `a(2,5)` has a particular **value** V. This may be regarded as saying that the defined relation `a` contains the tuple $(2,5,V)$. A ground term has at most one value, so this can be true for at most one V. The `+=` **aggregation operator** indicates that this V is obtained as $\sum_J$ `b(2,J) * c(J,5)`, where the summation is over all `J` for which `b(2,J)` and `c(J,5)` both have values. If there are no summands, then `a(2,5)` has no value, i.e., `a` contains no tuple of the form $(2,5,V)$. Other rules may use other aggregation operators. If `a(2,5)` unifies with the heads of

multiple rules, then its value is obtained by aggregating over the instantiated bodies contributed by all of those rules (all of which must have the same aggregation operator).

These and other features of Dyna make it possible to express many kinds of AI computations concisely and declaratively, as shown by Eisner and Filardo (2011). Their examples include graph algorithms such as shortest path and edit distance, finite and infinite neural networks, non-monotonic reasoning, constraint satisfaction via backtracking search and constraint propagation, exact parsing with probabilistic grammars, approximate inference in graphical models using message passing or Monte Carlo methods, and Markov decision processes. Each of their programs is essentially a system of Horn equations that must be solved to fixpoint. The use of variables in the rules makes it possible to specify a large system of equations with just a few rules. Fundamentally, the insight is that a computation can often be described using pattern-matching rules that describe the dependence of each named quantity on other named quantities.

Naive approaches to evaluating Dyna programs—the pure backward-chaining and forward-chaining strategies that are traditionally used in Prolog and Datalog respectively—often fail to terminate on real examples like the ones given by Eisner and Filardo (2011). This may occur either because of "infinite depth" (the chaining proceeds forever, continuing to visit new terms or cyclically revisit old ones) or because of "infinite breadth" (a single step of chaining touches infinitely many ground terms because a variable remains free).

We therefore propose a term-rewriting approach that is capable of rearranging computations over infinite relations and which can eliminate parts of a computation that are guaranteed not to affect the result. We will represent a Dyna program—or the intersection of a program with a query—using a finite algebraic object, namely an **R**-expr, which defines a set of key-value pairs. A step of execution then consists of rewriting this **R**-expr in a semantics-preserving way. We will define **R**-exprs and the rewrite rules in §§ 2 and 3.

In many cases, it is possible to eventually simplify the **R**-expr into a materialized finite relation. Even when the **R**-expr represents an infinite relation, it may still be possible to simplify it into a short expression that is readily interpretable by the human who issued the query (e.g., a materialized relation with don't-care variables, or a disjunction of conjunctions of arithmetic constraints). This "best-effort" behavior resembles that of a computer algebra system for simplifying mathematical expressions, or a constraint logic programming solver that may return answers with delayed constraints.

1.3 Two Dyna Examples

To illustrate why Dyna execution is difficult, we discuss two examples. The following program is formally quite similar to the previous matrix multiplication example. It defines all-pairs shortest paths in a directed graph. Each graph edge is defined by an `edge` term whose value is the length of that edge. For any `Start` and `End` such that there exists a directed path from `Start` to `End`, the value of `path(Start,End)` is the total length (via `+`) of the *shortest* such path (thanks to the aggregation operator `min=`).

```
path(Start,Start) min= 0.                                    % base case
path(Start,End) min= path(Start,Mid) + edge(Mid,End).        % recursive case
edge("baltimore", "washington dc") = 38.              % example rules for edges
edge("baltimore", "new york") = 195.
⋮                                                % many other edge rules omitted
```

This Dyna program is not a valid Datalog program, because the base case in line 1 establishes an infinite set of facts.[2] This could be remedied by changing line 1 to `path(Start,Start) min= 0 for city(Start)`, thus restricting `Start` to fall in a finite set of cities rather than ranging over all of $\mathcal{G}$. However, the original program is meaningful mathematically, so we would still like to be able to **query** the **path** relation that it defines—i.e., obtain a compact description of the set of all ground terms that match a given pattern, along with their values. (Terms with no value are not returned.) For example, the query `path(X,Y)` should return an infinite set; the query `path("atlantis",Y)` should return a finite set. Both of these sets contain `path("atlantis","atlantis")`.)

A Prolog-style solver does allow this program, and attempts to answer these queries by depth-first backward chaining that lazily instantiates variables with unification. A query's results may include non-ground terms such as `path(Start,Start)` that represent infinite sets. Unfortunately, the Prolog strategy will recurse infinitely on line 2, even on the query `path("atlantis",Y)`.

Yet `path("atlantis",Y)` is simply a single-source shortest path problem that *could* be answered by Dijkstra's 1959 algorithm. Similarly, the query `path(X,Y)` *could* be answered by running an all-pairs shortest path algorithm (e.g., Bellman-Ford) and augmenting the returned list of pairs with `path(Start,Start)`, which represents an infinite set of pairs. Our challenge is to construct a general Dyna engine that is capable of finding such algorithms.[3]

Our second example is a Dyna program that defines a convolutional neural network over any two-dimensional grayscale image. Larger images have more pixels; if there is a pixel at position `(X,Y)`, its intensity is given as the value of `pixel_brightness(X,Y)`. Lines 6–8 define a standard feed-forward network architecture in which each unit is named by a term in $\mathcal{G}$, such as `input(12,34)`, `hidden(14,31)`, or `output(kitten)`. The activation `out(J)` of unit `J` is normally computed as a sigmoided linear combination of the activations of other units `I`. The specific topology is given by the `edge(I,J)` items that connect `I` to `J`, which in this case are defined at lines 11–12. Specifically, `hidden(XX,YY)` is activated by the 9×9 square of neurons centered at `input(XX,YY)`. Then for each image property, `Property`, all of the hidden units are pooled to activate an output unit, `output(Property)`, whose activation represents the degree to which the image is predicted to have that property.

```
6  σ(X) = 1/(1+exp(-X)).                          % define sigmoid function for all X
7  in(J) += out(I) * edge(I,J).                                % vector-matrix product
8  out(J) += σ(in(J)).                                          % activation of node J
9  out(input(X,Y)) += pixel_brightness(X,Y)
       % extra external activation for input nodes
10 loss += (out(J) - target(J))**2.                    % L2 loss of the predictions
11 edge(input(X,Y),hidden(X+DX,Y+DY)) = weight_conv(DX,DY).                % layer 1
12 edge(hidden(XX,YY),output(Property)) = weight_output(Property).         % layer 2
13 weight_conv(DX,DY) := random(*,-1,1) for DX:-4..4, DY:-4..4.   % init with random
14 weight_output(Property) := random(*,-1,1).
```

For any given finite image, the program effectively defines a neural network with finitely many edges. Nonetheless, it is difficult to solve this system of equations using standard strategies. The difficulty arises from line 11. Using a forward chaining strategy (as is typical for Datalog), the existence of a value for `weight_conv(2,-3)` would forward-chain through line 11 to establish values (i.e., weights) for infinitely many edges of the form `edge(input(X,Y),hidden(XX,YY))` where

[2] The first version of Dyna (Eisner et al., 2005) also forbade this program, so as to allow a Datalog-like execution strategy.
[3] Ideally, it should also be able to handle infinite graphs that are defined by `edge` rules with variables, such as `edge(N,N+1) = 1`. Such a graph appears in the next example.

XX = X + 2 and YY = Y − 3 —even though only finitely many of these edges will turn out to be used when analyzing the finite image. Conversely, using backward chaining (as in prolog) to answer the (useful) query `out(output(kitten))` would lead to a subgoal query `edge(I,hidden(XX,YY))` with free variables XX and YY. That query must return the entire infinite set of input-to-hidden edges—even though for a given finite image, only finitely many of these edges will touch input units that actually have values.[4] These infinite sets in all of these examples cannot be represented by simple non-ground terms, because the variable arguments to `input`, `hidden`, and `weight_conv` in line 11 are related by arithmetic rather than simply by unification. However, our **R**-expr formalism makes it possible to keep track of these arithmetic constraints among the variables. §A describes how our approach handles this example program.

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** (Green, 2009) would be a map $\mathcal{G}^n \mapsto \mathcal{M}$ 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 named tuples $E : \mathcal{U} \mapsto \mathcal{G}$ 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, culminating in §4, which gives a scheme to translate a Dyna program into an **R**-expr.

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** given by the named tuple E. It defines a multiplicity $[\![\mathsf{R}]\!]_E$ 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 $E \mapsto [\![\mathsf{R}]\!]_E$. 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 $[\![\mathsf{R}]\!]_E$ depends only on the restriction of E to vars(R).

We say that T is a **term** if $\mathsf{T} \in \mathcal{V}$ or $\mathsf{T} = f(\mathsf{T}_1, \ldots, \mathsf{T}_n)$ where $f \in \mathcal{F}$ has rank n and $\mathsf{T}_1, \ldots, \mathsf{T}_n$ 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): $E(f(\mathsf{T}_1, \ldots, \mathsf{T}_n)) = f(E(\mathsf{T}_1), \ldots, E(\mathsf{T}_n))$.

We now define $[\![\mathsf{R}]\!]_E$ for each type of **R**-expr R, thus also presenting the different types of

[4] As another example, backward-chaining a query `edge(input(123,45),J))` to find the outgoing edges from a given input unit would lead to querying infinitely many weights of the form `weight_conv(DX,DY)` where DX = XX − 123 and DY = YY − 45—even though only finitely many of these weights will turn out to have values.

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 $\texttt{T},\texttt{U} \in \mathcal{T}$

Notice that we did not write $[\![\texttt{T}]\!]_E = [\![\texttt{U}]\!]_E$ but $E(\texttt{T}) = E(\texttt{U})$—since our denotation function $[\![\cdot]\!]_E$ maps **R**-exprs to multiplicities, but $E(\cdot)$ maps non-ground terms in $\mathcal{T}$ to ground terms in $\mathcal{G}$.

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 (Green, 2009):

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 $\texttt{A},\texttt{X} \in \mathcal{V}, \texttt{R} \in \mathcal{R}$. The $*$ in the summand means that we sum up $[\![\texttt{R}]\!]_{E[\texttt{X}=x]}$ copies of the value x (possibly ∞ copies). If there are no summands, the result of the $\sum \cdots$ is defined to be the identity element $\text{id}_{\texttt{sum}}$.

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 $\mathrm{id}_{\texttt{sum}} = 0$. In exactly the same way, one may define other aggregation operators such as `min`, based on the binary min operation and its identity element $\mathrm{id}_{\texttt{min}} = \infty$. Variants of these will be used in §4 to implement the aggregations in `+=` and `min=` rules like those in §1.3.

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[5] (unless they are bound by another operator within `R`). Thus 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. $[\![\texttt{M=count(R)}]\!]_E = \textbf{if } E(\texttt{M}) = [\![\texttt{R}]\!]_E \textbf{ then } 1 \textbf{ else } 0$

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 $\texttt{N} \notin \mathrm{vars}(\texttt{R})$), 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 $\texttt{R}_\texttt{f}$ with $\mathrm{vars}(\texttt{R}_\texttt{f}) \subseteq \{\texttt{X}_1, \ldots, \texttt{X}_n\}$ (which are n distinct variables) to serve as the **definition** (macro expansion) of `f`. Now define

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

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 (Eisner and Filardo, 2011). 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, so some queries against the **R**-expr may still yield simple answers.

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

3.1 Finite Materialized Relations

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

$$g = \begin{array}{|cc|}\hline \mathtt{X_1} & \mathtt{X_2} \\ \hline 1 & 1 \\ 2 & 6 \\ 2 & 7 \\ 2 & 7 \\ 5 & 7 \\ \hline \end{array} \xrightarrow{\text{to } \mathbf{R}\text{-expr}} \mathtt{R_g} = \begin{array}{lllll} (& (\mathtt{X_1}=1) & * & (\mathtt{X_2}=1) & \\ + & (\mathtt{X_1}=2) & * & (\mathtt{X_2}=6) & \\ + & (\mathtt{X_1}=2) & * & (\mathtt{X_2}=7) & \\ + & (\mathtt{X_1}=2) & * & (\mathtt{X_2}=7) & \\ + & (\mathtt{X_1}=5) & * & (\mathtt{X_2}=7) &) \end{array} \quad (1)$$

We see that each individual row of the table (tuple) translates into a product ($*$) of several ($\mathtt{Variable} = \mathtt{value}$) 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 $\mathtt{R_g}$ 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 $\mathtt{g(X_1,X_2)} \to \mathtt{R_g}$.

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 following structural unification rules provide the necessary rewrites:

$\mathtt{(}f\mathtt{(U_1,}\ldots\mathtt{,U_n)=}g\mathtt{(V_1,}\ldots\mathtt{,V_m))} \xrightarrow{1} \mathtt{0}$ **if** $f,g \in \mathcal{F}$ **and** $f \neq g$ ▷*functor clash (note that* $n \neq m \Rightarrow f \neq g$*)*

$\mathtt{(}f\mathtt{(U_1,}\ldots\mathtt{,V_n)=}f\mathtt{(V_1,}\ldots\mathtt{,V_n))} \xrightarrow{2} \mathtt{(U_1=V_1)} * \cdots * \mathtt{(U_n=V_n)}$ **if** $f \in \mathcal{F}$ **and** $n \geq 0$

$\mathtt{(T=X)} \xrightarrow{3} \mathtt{(X=T)}$ **if** $\mathtt{X} \in \mathcal{V}$ **and** $\mathtt{T} \in \mathcal{T}$ ▷*put var on left to match rules below*

$\mathtt{(X=X)} \xrightarrow{4} \mathtt{1}$ **if** $\mathtt{X} \in \mathcal{V}$ ▷*because true for every value of* `X`

$\mathtt{(X=T)} \xrightarrow{5} \mathtt{0}$ **if** $\mathtt{X} \in \mathcal{V}$ **and** $\mathtt{T} \in \mathcal{T}$ **and** $\mathtt{X} \in \mathrm{vars}(\mathtt{T})$ ▷*not true for any* `X` *(occurs check)*

We now have an arithmetic expression `0*0+1*0+1*1+1*1+0*1`, which we can reduce to the multiplicity `2` via rewrites that implement basic arithmetic on multiplicities $\mathcal{M}$:

$\mathtt{M} + \mathtt{N} \xrightarrow{6} \mathtt{L}$ **if** $\mathtt{M},\mathtt{N} \in \mathcal{M}$ **and** $\mathtt{M}+\mathtt{N}=\mathtt{L}$; $\quad \mathtt{M} * \mathtt{N} \xrightarrow{7} \mathtt{L}$ **if** $\mathtt{M},\mathtt{N} \in \mathcal{M}$ **and** $\mathtt{M}*\mathtt{N}=\mathtt{L}$

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 $\texttt{R}_\texttt{g}\texttt{*(X}_1\texttt{=2)*(X}_2\texttt{=7)}$. This filters the original relation to just the pairs that match $(2,7)$, and simplifies to $\texttt{2*(X}_1\texttt{=2)*(X}_2\texttt{=7)}$. To accomplish this simplification, we need to use the following crucial rewrite rule:

$$\texttt{(X=T)*R} \xrightarrow{8} \texttt{(X=T)*R}\{\texttt{X} \mapsto \texttt{T}\} \textbf{ if } \texttt{X} \in \mathcal{V} \textbf{ and } \texttt{T} \in \mathcal{T} \textbf{ and } \texttt{X} \in \text{vars}(\texttt{R}) \qquad \triangleright \textit{equality propagation}$$

As a more interesting example, the reader may consider simplifying the query $\texttt{R}_\texttt{g}\texttt{*lessthan(X}_2\texttt{,7)}$, 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 $\texttt{f(X}_1\texttt{,X}_2\texttt{)} \to \texttt{R}_\texttt{f}$.

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

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.[6] 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 $\texttt{Q},\texttt{R},\texttt{S} \in \mathcal{R}$:

$$\texttt{1 * R} \xrightarrow{9} \texttt{R} \qquad \triangleright \textit{multiplicative identity}$$

$$\texttt{0 * R} \xrightarrow{10} \texttt{0} \qquad \triangleright \textit{multiplicative annihilation}$$

$$\texttt{0 + R} \xrightarrow{11} \texttt{R} \qquad \triangleright \textit{additive identity}$$

$$\infty \texttt{ * R} \xrightarrow{12} \infty \textbf{ if } \texttt{R} \in \mathcal{M} \textbf{ and } \texttt{R} > \texttt{0} \qquad \triangleright \textit{absorbing element}$$

$$\infty \texttt{ + R} \xrightarrow{13} \infty \qquad \triangleright \textit{absorbing element}$$

$$\texttt{R + S} \xleftrightarrow{14} \texttt{S + R}; \qquad \texttt{R * S} \xleftrightarrow{15} \texttt{S * R} \qquad \triangleright \textit{commutativity}$$

$$\texttt{Q + (R + S)} \xleftrightarrow{16} \texttt{(Q + R) + S}; \qquad \texttt{Q * (R * S)} \xleftrightarrow{17} \texttt{(Q * R) * S} \qquad \triangleright \textit{associativity}$$

$$\texttt{Q * (R + S)} \xleftrightarrow{18} \texttt{Q * R + Q * S} \qquad \triangleright \textit{distributivity}$$

$$\texttt{R * M} \xrightarrow{19} \texttt{R + (R * N)} \textbf{ if } \texttt{M,N} \in \mathcal{M} \textbf{ and } \texttt{(1+N=M)} \qquad \triangleright \textit{implicitly does } \texttt{M} \to \texttt{1+N}$$

$$\texttt{R} \xleftrightarrow{20} \texttt{R * R} \textbf{ if } \texttt{R} \text{ is a constraint} \qquad \triangleright \textit{as defined in §2}$$

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

[6] 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.

```
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 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 restricts a prefix such as I or (I,J), 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,[7] 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 (Frühwirth, 1998). 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 cases:

$$\texttt{plus(I,J,K)} \xrightarrow{21} \underbrace{\mathbb{I}(\texttt{I}+\texttt{J}=\texttt{K})}_{\in\{0,1\}} \textbf{ if } \texttt{I},\texttt{J},\texttt{K} \in \mathbb{R}$$

$$\texttt{plus(I,J,X)} \xrightarrow{22} \texttt{(X=I+J)} \textbf{ if } \texttt{I},\texttt{J} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(I,X,K)} \xrightarrow{23} \texttt{(X=K-I)} \textbf{ if } \texttt{I},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

$$\texttt{plus(X,J,K)} \xrightarrow{24} \texttt{(X=}\underbrace{\texttt{K}-\texttt{J}}_{\in\mathbb{R}}\texttt{)} \textbf{ if } \texttt{J},\texttt{K} \in \mathbb{R} \textbf{ and } \texttt{X} \in \mathcal{V}$$

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).

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

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 (Frühwirth, 1998), 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 idempotent **constraint propagators**. Other useful examples concerning `plus` include `plus(0,J,K)` $\xrightarrow{25}$ `K=J` and `plus(I,J,J)` $\xrightarrow{26}$ `(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)` $\xrightarrow{27}$ `0`; the transitivity rule `lessthan(I,J)*lessthan(J,K)` $\xrightarrow{28}$ `lessthan(I,J)*lessthan(J,K)*lessthan(I,K)`; and `lessthan(0,I)*plus(I,J,K)` $\xrightarrow{29}$ `lessthan(0,I)*plus(I,J,K)*lessthan(J,K)`. The integer domain can be **split** by rules such as `int(I)` $\xrightarrow{30}$ `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` $\xrightarrow{31}$ `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)` $\xrightarrow{32}{}^{*}$ `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)` $\xleftrightarrow{33}$ `proj(X,R) + proj(X,S)` ▷*distributivity over* +

`proj(X,R*S)` $\xleftrightarrow{34}$ `R*proj(X,S)` **if** `X` $\notin$ 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))` $\xrightarrow{35}$ 1 **if** `T` $\in \mathcal{T}$ **and** `X` $\notin$ vars(`T`) ▷*occurs check*

`proj(A,(A=sum(X,R)))` $\xrightarrow{36}$ 1 **if** `A` $\notin$ vars(`R`) ▷*cardinality of an aggregated variable*

`proj(X,R)` $\xrightarrow{37}$ `R*∞` **if** `X` $\notin$ vars(`R`) ▷*cardinality of an unconstrained variable*

`proj(X,bool(X))` $\xrightarrow{38}$ 2 ▷*cardinality of a variable given a certain unary constraint*

`proj(X,int(X))` $\xrightarrow{39}$ ∞ ▷*cardinality of a variable given a certain unary constraint*

`proj(X,proj(Y,nand(X,Y)))` $\xrightarrow{40}$ 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`:

$$\texttt{M=count(R+S)} \xrightarrow{41} \texttt{proj(L,(L=count(R)) * proj(N,(N=count(S)) * plus(L,N,M)))}$$

$$\texttt{M=count(R*S)} \xrightarrow{42} \texttt{proj(L,(L=count(R)) * proj(N,(N=count(S)) * times(L,N,M)))} \textbf{ if } \mathrm{vars}(\texttt{R}) \cap \mathrm{vars}(\texttt{S}) = \emptyset$$

$$\texttt{M=count(N)} \xrightarrow{43} \texttt{(M=N)} \textbf{ if } \texttt{N} \in \mathcal{M}$$

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`.)

$$\texttt{A=sum(X, 0)} \xrightarrow{44} \texttt{(A=id}_{\texttt{sum}}\texttt{)}$$

$$\texttt{A=sum(X, (X=T))} \xrightarrow{45} \texttt{(A=T)} \textbf{ if } \texttt{T} \in \mathcal{T} \textbf{ and } \texttt{X} \notin \mathrm{vars}(\texttt{T}) \qquad \triangleright \textit{occurs check}$$

$$\texttt{A=sum(X, R+S)} \xrightarrow{46} \texttt{proj(B, (B=sum(X,R)) * proj(C, (C=sum(X,S)) * plus(B,C,A)))}$$

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:[8]

$$\texttt{A=sum(X, (X=sum(Y,R)))} \xrightarrow{47} \texttt{(A=sum(Y,R))} \textbf{ if } \texttt{X} \notin \mathrm{vars}(\texttt{R})$$

$$\texttt{A=sum(X, (X=min(Y,R)))} \xrightarrow{48} \texttt{(A=min(Y,R))} \textbf{ if } \texttt{X} \notin \mathrm{vars}(\texttt{R})$$

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:

$$\texttt{A=sum(X,R*S)} \xrightarrow{49} \texttt{proj(B, sum\_copies(R,B,A) * (B=sum(X,S)))} \quad \textbf{if } \texttt{X} \notin \mathrm{vars}(\texttt{R})$$

[8] 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.

Here, `sum_copies(M,B,A)` for $\texttt{M} \in \mathcal{M}$ constrains `A` to be the aggregation of $\texttt{M} \in \mathcal{M}$ 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

$$\begin{aligned}
[\![\texttt{sum_copies(R,B,A)}]\!]_E &= \textbf{if } [\![\texttt{R}]\!]_E \cdot [\![\texttt{B}]\!]_E = [\![\texttt{A}]\!]_E \textbf{ then } 1 \textbf{ else } 0 \\
[\![\texttt{min_copies(R,B,A)}]\!]_E &= \textbf{if } ([\![\texttt{R}]\!]_E = 0 \textbf{ and } [\![\texttt{A}]\!]_E = \mathsf{id}_{\texttt{min}}) \\
&\quad \textbf{or } ([\![\texttt{R}]\!]_E > 0 \textbf{ and } [\![\texttt{A}]\!]_E = [\![\texttt{B}]\!]_E) \textbf{ then } 1 \textbf{ else } 0
\end{aligned}$$

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

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

$$\texttt{sum\_copies(R,B,A)} \xrightarrow{50} \texttt{proj(M,(M=count(R))*times(M,B,A))} \qquad \triangleright \textit{assumes } \mathsf{id}_{\texttt{sum}} = 0$$

$$\texttt{min\_copies(R,B,A)} \xrightarrow{51} \texttt{proj(M,(M=count(R))*((M=0)*(A=id}_{\texttt{min}}\texttt{)+lessthan(0,M)*(A=B)))}$$

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 $\texttt{I} \in \text{vars}(\texttt{R}), \texttt{J} \in \text{vars}(\texttt{S}), \texttt{K} \notin \text{vars}(\texttt{R*S})$, and $\text{vars}(\texttt{R}) \cap \text{vars}(\texttt{S}) = \emptyset$. 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)` $\xrightarrow{52}$ `(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 $\geq k$:

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

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)` $\xrightarrow{53}$ `(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 Dyna programs 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.

```
15 f(X) += X*X.
16 f(4) += 3.
17 f(X) += g(X,Y).
18 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`.[9]

Dyna mandates in-place evaluation of Dyna expressions that have values (Eisner and Filardo, 2011). 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 each contribute one `Result`, while the third rule contributes as many `Result`s as the map has keys of the form `g(4,Y)`.[10]

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` built-in, as described before, with identity element $\mathsf{id}_{\texttt{sum}} = 0$. There is a complication, however: in Dyna, a `Key` that has 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

[9] 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.

[10] 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`.

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 10 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) --54--> 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`.[11]

A Dyna program may consist of multiple *dynabases* (Eisner and Filardo, 2011). 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.

5 Related Work

Dyna is hardly the first programming language—or even the first logic programming language—to model its semantics in terms of relational expressions or to use term rewriting for execution.

Relational algebras provide the foundation of relational databases (Codd, 1970). We require an extension from set relations to bag relations (cf. Green, 2009) in order to support aggregation, as aggregation considers the multiplicity of bag elements. Our particular formulation in §2 uses

[11] 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.

variable names rather than column indices; this allows us to construct and rearrange **R**-exprs without the technical annoyance of having to track and modify column indices.

Datalog has long been understood as an alternative mechanism for defining the sort of finite relations that are supported by relational databases (Ceri et al., 1989; Gallaire et al., 1984). Indeed, it is possible to translate Datalog programs into a relational algebra that includes a fixpoint operator. Bellia and Occhiuto (1990) and Arias et al. (2017) generalize some of these translation constructions to pure logic programming, though without aggregation or built-ins. To attain the full power of relational algebras, Datalog is often extended to support aggregation and negation (using "stratified" programs to control the interaction of these operations with recursion). Implementations of Datalog often use a forward-chaining (or "semi-naive bottom-up") strategy that materializes all relations in the user-defined program (Ullman, 1988), and this was used in the first version of Dyna (Eisner et al., 2005).

Materialization is a sort of brute-force strategy that is guaranteed to terminate (for stratified programs) when all relations are finite, but which is not practical for very large or infinite relations. Thus, our goal in this paper was to make it possible to manipulate intensional descriptions of possibly infinite relations. The work that is closest to our approach is constraint logic programming (Jaffar and Maher, 1994), which extends Prolog's backtracking execution (SLD resolution) so that the environment in which a subgoal is evaluated includes not only variable bindings from Prolog unification, but also a "store" of **delayed constraints** (so called because they have not yet been rewritten into simple unification bindings as would be required by SLD resolution). This conjunction of unification binding constraints and delayed constraints on a common set of variables is continually rewritten using constraint handling rules (Frühwirth, 1998) such as simplification and propagation. Our pseudocode in §B.2 shares this notion of an environment with constraint logic programming, and applies our rewrite rules from §3 in essentially the same way. One difference is that our pseudocode is breadth-first—it rewrites disjuncts in parallel rather than backtracking. This increases the memory footprint, but sometimes makes it possible to short-circuit an infinite regress that is provably irrelevant to the final answer. (Our rewrite rules could be applied in any order, of course.) Another difference is that our **R**-expr formalism supports the construction of new relations from old ones by aggregation and projection, as required by the Dyna language. To our knowledge, these extensions to expressivity have not been proposed within constraint logic programming (although Ross et al. (1998) do consider aggregation constraints on first-class bag-valued variables).

Of course, functional programming is also based on term rewriting (e.g., reduction strategies for the λ-calculus), and term rewriting has figured in attempts to combine functional and logic programming. The Curry language (Hanus, 1997, 2007) rewrites terms that denote computable functions, but supports the Prolog-like ability to call a function on variables that have not yet been bound. This motivates a rewriting technique called narrowing (Antoy, 2010) that can result in nondeterministic computation (e.g., backtracking) and is related to unfolding (inlining) in logic programming. Whereas Curry simplifies terms that denote arbitrary functions (potentially higher-order functions), our proposed system for Dyna rewrites terms that specifically denote bag relations (i.e., functions from ground tuples to multiplicities). We remark that Curry appears to have committed to a specific rewrite ordering strategy resembling Haskell's fixed execution order. While we give a specific strategy for rewriting Dyna in §B.2, we have also explored the possibility of *learning* execution strategies for Dyna that tend to terminate quickly on a given workload (Vieira et al., 2017).

6 Conclusion

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" (Vieira et al., 2017) by constructing or discovering reusable strategies for reducing expressions of frequently encountered sorts (i.e., polyvariant specialization, including memoization, together with programmable rewrite strategies).

References

ANTOY, S. 2010. Programming with narrowing: A tutorial. *Journal of Symbolic Computation 45,* 5, 501 – 522. Symbolic Computation in Software Science.

ARIAS, E. J. G., LIPTON, J., AND MARIÑO, J. 2017. Constraint logic programming with a relational machine.

BELLIA, M. AND OCCHIUTO, M. E. 1990. C-expressions: a variable-free calculus for equational logic programming.

CERI, S., GOTTLOB, G., AND TANCA, L. 1989. What you always wanted to know about Datalog (and never dared to ask). In *IEEE Transactions on Knowledge and Data Engineering*.

CLOCKSIN, W. F. AND MELLISH, C. S. 1984. *Programming in Prolog*. Springer-Verlag.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM 13,* 6.

COLMERAUER, A. AND ROUSSEL, P. 1996. *The Birth of Prolog*. Association for Computing Machinery, Chapter 7.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

EISNER, J. AND FILARDO, N. W. 2011. Dyna: Extending Datalog for modern AI. In *Datalog Reloaded*, O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Lecture Notes in Computer Science. Springer.

EISNER, J., GOLDLUST, E., AND SMITH, N. A. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*. Vancouver, 281–290.

FRÜHWIRTH, T. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming 37,* 1.

FRÜHWIRTH, T. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming 37,* 1.

GALLAIRE, H., MINKER, J., AND NICOLAS, J.-M. 1984. Logic and databases: A deductive approach. *ACM Comput. Surv. 16,* 2.

GREEN, T. J. 2009. Bag semantics. In *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Springer.

HANUS, M. 1997. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. ACM, New York, NY, USA.

HANUS, M. 2007. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*. Springer LNCS 4670, 45–75.

JAFFAR, J. AND MAHER, M. J. 1994. Constraint Logic Programming: A survey. *J. Logic Program. 19,20*, 503–581.

ROSS, K. A., SRIVASTAVA, D., STUCKEY, P. J., AND SUDARSHAN, S. 1998. Foundations of aggregation constraints. *Theoretical Computer Science 193,* 1, 149–179.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Vol. 1. Computer Science Press.

VIEIRA, T., FRANCIS-LANDAU, M., FILARDO, N. W., KHORASANI, F., AND EISNER, J. 2017. Dyna: Toward a self-optimizing declarative language for machine learning applications. In *Proc. of the ACM SIGPLAN Workshop on Machine Learning and Programming Languages*.

A How R-exprs Manage Infinite Relations

Here we illustrate how to successfully simplify **R**-exprs that involve infinite relations, as in our two examples of §1.3: all-pairs shortest path and convolutional neural networks. We focus on how our term rewriting approach is able to handle the neural network example (lines 6–14). The shortest-path example succeeds for similar reasons, and we leave it as an exercise for the reader.[12]

Recall that the neural network defined in §1.3 defines an *infinite* number of `edge`s between input units and hidden units, not all of which are used in the case of a finite image.[13] To be able to successfully answer queries against this program, we must avoid materializing the infinite `edge` relation.

First, we translate the program at lines 6–14 Dyna into a user-defined constraint, `is(Key,Val)` $\rightarrow \cdots$. For readability, we show only the specialization of this constraint to cases where `Key` is an `edge`:

```
is(edge(Arg1,Arg2),Result) → proj(X, proj(Y, proj(DX, proj(DY, proj(XX, proj(YY,
    (Arg1=input(X,Y))*(Arg2=hidden(XX, YY))*     ▷the nested Key pattern has been decomposed
                                                    ▷into equality constraints
    plus(X,DX,XX)*plus(Y,DY,YY)*              ▷arithmetic + in Dyna becomes plus in the R-expr;
                                          ▷new variables XX and YY are introduced for the results
    is(weight_conv(DX, DY), Result)       ▷recurse to another branch of the is constraint (below)
)))))) + proj(XX, proj(YY, proj(Property,
  (Arg1=hidden(XX, YY))*(Arg2=output(Property))*
  is(weight_output(Property), Result)     ▷recurse to another branch of the is constraint (below)
)))
```

The first summand describes the input-to-hidden edges, and the second summand describes the hidden-to-output edges. The specializations of the user-defined constraint to the weight parameters would look like this (recall that the parameters were set randomly):

```
is(weight_conv(DX, DY), Result) → (                  ▷weight_conv is a table similar to §3.1
  (DX=-4)*(DY=-4)*(Result = .123) +
  (DX=-4)*(DY=-3)*(Result = .173) +
  (DX=-4)*(DY=-2)*(Result = .281) +
  ⋮
  (DX=4)*(DY=4)*(Result = .971) )

is(weight_output(Property), Result) → (                          ▷weight_output table
  (Property=kitten)*(Result=.777) +
  (Property=puppy)*(Result=.643) +
  ⋮
  (Property=zebra)*(Result=.912) )
```

The right-hand side of the first rule can be further rewritten by replacing the call to `weight_conv` with its definition. Then we can use the rewrite rules from §3.5 that to lift the resulting disjunction out of the scope of the `proj` operator. This gives us the following answer to the query `is(edge(Arg1, Arg2), Result)`, which cannot be further simplified:

[12] The answer appears on our talk slides from WRLA 2020 (see footnote 1).

[13] Our approach can also attempt to answer queries even when the image is an infinite image that is specified by rule, such as an all-white image or a tessellation of the plane. Depending on the specification of the image, the resulting **R**-expr may or may not simplify to yield a simple numerical answer for (say) the value of `out(output(kitten))`. We do not consider such cases in this appendix.

```
is(edge(Arg1, Arg2), Result) →* proj(X, proj(Y, proj(XX, proj(YY,
    (Arg1=input(X,Y))*(Arg2=hidden(XX,YY))*
    plus(X,-4,XX)*plus(Y,-4,YY)*
    (Result=.123)
    )))) + proj(X, proj(Y, proj(XX, proj(YY,
    (Arg1=input(X,Y))*(Arg2=hidden(XX,YY))*
    plus(X,-4,XX)*plus(Y,-3,YY)*
    (Result=.173)
    )))) + ···                          ▷Additional branches of the disjunction omitted
```

This answer still defines infinitely many edges, where the variables `X` and `XX` (similarly `Y` and `YY`) are related by `plus` constraints.

If we were rewriting the above **R**-expr in the context of a larger computation such as a query for `out(output(kitten))` on a particular image, then we would be able to specialize it further against the finite set of input pixels. In this case, suppose that only a single input unit's activation `out(input(0,0))` appears as a key in the `is` relation. Further simplification of the query will result in a sub-**R**-expr `is(edge(Arg1, Arg2), Result)*(Arg1=input(0,0))`, which effectively specializes `Arg1` to this unit. This will eliminate the disjunctive branches of `weight_output` and set the values of `X=0` and `Y=0` for all of the `weight_conv` rules. This will then allow for all of the `plus(X,-4,XX)` and `plus(Y,-4,YY)` expressions above to run. Further propagation of the values of these constraints will eventually determine the value of `Arg2`, identifying the finite number of edges which are actually required to run this expression:

```
is(edge(Arg1,Arg2),Result)*(Arg1=input(0,0)) → (
  (Arg2=output(-4,-4))*(Result=.123) +
  (Arg2=output(-4,-3))*(Result=.173) +
  ··· )
```

B R-expr Solver Implementation Details

B.1 Simplification in an environment

Our rewriting procedure is based around *simplifying* a **R**-expr until it reaches a normal form (meaning that there are no more rewrites which can be applied).

Scanning an entire **R**-expr and identifying all available rewrites could be expensive if we were to directly apply the rules presented in this paper. One reason is that many of the rules can only apply once two distant constraints have been brought together into a conjunction `R*S`. For example, a reduction like `(X=2)*···*(plus(X,3,Y)+(Z=9))` →* `(X=2)*···*((Y=5)+(Z=9)` requires quite a lot of rearrangement to create a copy of `X=2` (using the rewrite `R` → `R*R`), move this copy rightward (using associative and commutative rewrites), and push it down into the disjuncts (using a distributive rewrite), finally obtaining a subterm `(X=2)*plus(X,3,Y)` that can be rewritten as `(X=2)*(Y=5)`. Now we have `(X=2)* ···*((X=2)*(Y=5)+(X=2)*(Z=9))`, and many of these steps must be carried out in reverse to consolidate the extra copies of `X=2` back into the original.

Our top-down approach will instead implicitly percolate constraints to where they are needed by maintaining an **environment**—a bag of currently available constraints conjunctive, essentially the same as the constraint store used in constraint logic programming. In the example above, when the rewriting engine encounters `plus(X,3,Y)`, it will be able to look up the value of `X` in the current environment $\mathcal{C}$. This environment is passed by reference through the operational semantics pseudocode (below) in much the same way as the environment of §2 was threaded through the

denotational semantics definitions, although the two notions of environment are distinct and have different types.

An important benefit of using an environment is that we can eliminate reversible rules that were presented in the main paper (associativity, commutativity, distributivity, constraint duplication), whose reversibility prevented the existence of normal forms. In effect, we now have a composite rule something like "rewrite `plus(X,3,Y)` → `Y=5` provided that `X=2` must also be true," which implicitly invokes whatever associative/commutative/distributive rewrites are necessary to percolate a copy of `X=2` to the position where it can license this composite rule. We need not explicitly invoke the reversible rules.

We define a SIMPLIFY operator which takes a **R**-expr `R` and returns its normal form. The operator is also given the current environment as a second argument, as a place to find and store unordered conjunctive constraints outside of `R` itself.

Internally, SIMPLIFY makes use of two operators SIMPLIFYFAST and SIMPLIFYCOMPLETE. SIMPLIFYFAST allows the system to quickly identify rewrites that are useful, but which require a *minimal* amount of work to identify. It works by using the environment to track ground assignments to variables. This allows a constraint like `plus(X,3,Y)` to quickly check the environment to see whether `X` or `Y` is currently bound to a number without having to scan or modify the entire **R**-expr. SIMPLIFYCOMPLETE is similar, but it identifies *all* possible rewrites that can currently be applied (not just the fast ones). Thus, it uses an enhanced environment that can also track other types of constraints, similar to the constraint store in ECLiPSe. The enhanced environment may now include structural equality constraints on non-ground terms, other built-in constraints such as `plus` and `lessthan`, user-defined constraints, and even constraints such as `proj(X, ···)` and `A=sum(···)`. This allows the system to rapidly identify the contradiction in an expression such as `lessthan(A,B)*···*lessthan(B,A)` →`0` without using explicit associative or commutative rewrites. The enhanced environment does not include disjunctive constraints of the form `R+S`.

B.2 Pseudocode of SIMPLIFY

To simplify an **R**-expr to a normal form, the system starts by applying SIMPLIFYFAST repeatedly until it is unable to make any more changes. The system will then try a single application of SIMPLIFYCOMPLETE in case there are any additional rewrites that can be applied at this point. Note: the entry point for SIMPLIFYCOMPLETE is SIMPLIFYCOMPLETEMAIN. If so, it returns to SIMPLIFYFAST and repeats the process.

```
1:  function SIMPLIFY(R)
2:      C ← ∅                                   ▷ Construct an empty environment to start
3:      R ← UNIQIFYVARS(R)                      ▷ Ensure that all of the variables in the R-expr have unique names
4:      loop                                    ▷ Outer loop tries SIMPLIFYCOMPLETE only when there are no more fast rewrites available.
5:          loop                                ▷ Inner loop performs faster rewrites using SIMPLIFYFAST
6:              R' ← SIMPLIFYFAST(R, C)
7:              if R == R' : break              ▷ SIMPLIFYFAST has finished rewriting
8:              R ← R'
9:          R' ← SIMPLIFYCOMPLETEMAIN(R, C)
10:         if R == R' : break                  ▷ SIMPLIFYCOMPLETE has finished rewriting
11:         R ← R'
12:     return C * R                            ▷ The R-expr has reached a normal form, return to caller
```

UNIQIFYVARS(R) is a preprocessing step that ensures that distinct variables within R have unique names. In practice, it accomplishes this by giving them distinct integer subscripts. In particular, in an operator sub-expression of the form `proj(X,S)` or `sum(X,S)`, an unused integer i is chosen, and the first argument X as well as all copies of X that are bound by it are replaced with X_i.

The invariant of having unique names will be preserved by the rest of the SIMPLIFY procedure. In particular, when a rewrite rule introduces new variables (e.g., when expanding an aggregation or inlining a user-defined constraint), these variables are assigned unused names. In practice, the variable names are augmented with unused integers as subscripts.

An environment $\mathcal{C}$ consists primarily of an array or hash map that maps key i to a list of the constraints in $\mathcal{C}$ that constrain the variable subscripted with i. Thanks to the unique-name invariant, distinct variables such as X_5 and X_8 can appear in the same environment.

SIMPLIFYFAST performs some simple rewrites in a single pass over the **R**-expr, matching against nodes of the **R**-expr syntax tree. The pseudocode shown here is incomplete (it omits some simple rewrites that are in fact handled by our SIMPLIFYFAST, such as multiplicity arithmetic from §3.2). However, it gives the general flavor of how rewriting with an environment works. The environment is passed by reference and is destructively modified. At the top level, SIMPLIFYFAST is called on R and an empty environment $\mathcal{C}$. It returns R' and destructively adds some constraints into $\mathcal{C}$, which are no longer necessarily enforced by R'. Thus, R is denotationally equivalent to $\mathcal{C}$ *R' (where $\mathcal{C}$ is interpreted as the product of the constraints in $\mathcal{C}$), and R $\rightarrow^*$ $\mathcal{C}$*R' under the rewrite rules of the main paper.

```
1:  function SIMPLIFYFAST(R, C)
2:      if R matches (X=x) with x ∈ G : ▷ Handle equality constraints (with ground values). rewrite rule 8
3:          if X ∈ C :                          ▷ Check if X's value in C matches the constraint x
4:              if C[X] == x :        ▷ We use "C[X]" as a shorthand for "(X=x) ∈ C" that returns "x"
5:                  return 1                                        ▷ Successful unification
6:              else
7:                  return 0
8:          else
9:              C[X] ← x                                            ▷ C is modified in place
10:             return 1
11:     else if R matches (X=Y) and Y ∈ C :
12:         ▷ Same as lines 3 through 10, but setting x =C[Y]
13:     else if R matches (X=X) :
14:         return 1                                                ▷ Vacuously true
15:     else if R matches (X=f(U_1, ..., U_n)) : ▷ Structural unification between ground terms. rewrite rules 1
    and 2
16:         if X ∈ C :                                  ▷ In the case that X is a ground value
17:             x ← C[X]
18:             return (U_1=x_1)*···*(U_n=x_n)          ▷ Unify all of the arguments of the structure
19:         else if U_1 ∈ C and ··· and U_n ∈ C :                   ▷ All of U_i are ground values
20:             u_1 ← C[U_1] ; ··· ; u_n ← C[U_n]
21:             return (X=f(u_1, ..., u_n))             ▷ Construct a term from the values of u_i
22:         else
23:             return R                    ▷ Neither side is ground, no change in the returned value
24:     else if R matches Q*S :                                     ▷ Conjunctive R-exprs
```

```
25:        Q' ← SimplifyFast(Q, C)
26:        if Q' matches 0 : return 0          ▷ The first conjunct has failed, so it can stop processing early.
   Implements rewrite rule 10
27:        S' ← SimplifyFast(S, C)
28:        if S' matches 0 : return 0          ▷ Implements rewrite rule 10
29:        if Q' matches 1 : return S'         ▷ Implements rewrite rule 9
30:        if S' matches 1 : return Q'         ▷ Implements rewrite rule 9
31:        if S' ∈ M and Q' ∈ M :              ▷ Product of multiplicities as in rewrite rule 7
32:            L ← S' * Q'
33:            return L
34:        return Q'*S'
35:    else if R matches Q+S :                 ▷ Disjunctive R-exprs
36:        ▷ Below, we copy the environment so that it can be modified separately by each recursive call. These could
            be lazy copies (copy on write), or managed via a call stack with undo operations as in Prolog.
37:        C_Q ← Copy(C)
38:        C_S ← Copy(C)
39:        Q' ← SimplifyFast(Q, C_Q)
40:        S' ← SimplifyFast(S, C_S)
41:        if Q' matches 0 :                   ▷ Remove Q' as it is now an empty branch
42:            C ← C_S                         ▷ Copy the environment of non-empty branch
43:            return S'
44:        if S' matches 0 :
45:            C ← C_Q
46:            return Q'
47:        ▷ Below, any constraints that are shared between C_Q and C_S are propagated to C. Recall that SimplifyFast
            's environment only contains ground variable assignments, so this is only checking equalities
            between the same key for the variables.
48:        C ∪= C_Q ∩ C_S                      ▷ Implement rewrite rule 18 for ground assignments only
49:        ▷ Anything in C_Q and C_S that could not be merged into C is added back to the R-expr returned.
50:        if Q' ∈ M and S' ∈ M and C_Q − C = ∅ and C_S − C = ∅ :
51:            L ← Q' + S'                     ▷ Both Q' and S' have the same environment and are multiplicities,
52:            return L                        ▷ so we can merge the R-exprs (add the multiplicities)
53:        return (C_Q − C) * Q' + (C_S − C) * S'
54:    else if R matches proj(X, Q) :
55:        ▷ Omitted: Rewrites from §3.5 are applied first, otherwise pass through to the nested R-expr below.
56:        ...
57:        ▷ Note: Since variable are uniqified, the system does not have to worry about clobbering the slot of another
            variable with the same name in the surface syntax.
58:        Q' ← SimplifyFast(Q, C)
59:        Q' ← (X=C[X])*Q'                    ▷ Save the value of the variable X into the R-expr
60:        return proj(X, Q')
61:    else if R matches (A=sum(X, Q)) :       ▷ Match aggregators, exemplified here by sum
62:        ▷ Omitted: The rules described in §3.6 for aggregation are run first.
63:        ...
64:        ▷ In the case that none of those rewrites match, the body of the aggregator is matched similar to the
            projection case.
```

```
65:       Q' ← SIMPLIFYFAST(Q, C)
66:       Q' ← (X=C[X])*Q'
67:       return (A=sum(X, Q'))
68:   else if R matches (A=count(Q)) :
69:       ▷ Omitted: The rules described in §3.6 for the count expression
70:       ...
71:       Q' ← SIMPLIFYFAST(Q, C)                    ▷ If no rules match, simplify the inner expression
72:       if Q' ∈ M :                                ▷ Q' is a multiplicity, assign its value to A
73:           return (A=Q')
74:       return (A=count(Q'))
75:   else if R matches is(f(T_1,..., T_n)^m,F) :                          ▷ rewrite rule 54
76:       ▷ We prevent unbounded amounts of recursive inlining. We track the number of times a rule has be applied.
          If the recursion limit is exceeded, we simply refuse to rewrite further.
77:       if m > RECURSIONLIMIT : return R
78:       ▷ The code below, implements a single step of inlining. We lookup a R-expr by name corresponding to a
          user-defined rule. We replace placeholder variables X_i with the variables at this call site, and uniqify
          variables within the expression.
79:       R' ← LOOKUPNAMED(f)
80:       R' ← R' {X_1 ↦ T_1, ··· , X_n ↦ T_n}
81:       R' ← UNIQIFYVARS(R')
82:       R' ← R' {f(···)^0 → f(···)^(m+1)}          ▷ All calls are annotated with the depth m+1
83:       return R'
84:   else if R matches plus(I,J,K) :
85:       ▷ The rules for plus are equivalent to rewrite rules 21 to 24 given in §3.4
86:       if I ∈ C and J ∈ C :
87:           return (K=C[I] + C[J])                 ▷ Compute the addition and set K to the result.
88:       else if I ∈ C and K ∈ C :
89:           ...                                    ▷ Other modes similar to the above, omitted for brevity
90:       else
91:           return R                               ▷ Return unchanged in the case that it can not run
92:   else if R matches ··· :
93:       ▷ Omitted: Other matching rules are very similar to the rewrites presented in the main paper.
94:       ...
95:   else
96:       return R                  ▷ In the case that no rewrite matches, then the original R-expr is returned
```

The main purposes of SIMPLIFYFAST are to handle simple rewrites including evaluating built-in operations, expanding aggregations and function calls by one step, and carrying out equality propagation (§3.2). If we directly used the rewrite system of the main paper, then we would have to deliberately apply commutative, associative, and distributive rewrites to obtain a subterm of the form to which the equality propagation rewrite applies.

Notice above that equality constraints are *absorbed* into the environment. When one of these equalities is dropped from the environment (line 9), it is restored as a conjunct at the front of the **R**-expr (lines 48 and 53). Since SIMPLIFYFAST traverses the **R**-expr from left to right, this rearrangement ensures that the next time it is called, the environment will get fully populated with these equality constraints before SIMPLIFYFAST moves rightward to rewrite other constraints in the expression that might need to look up the values of variables in the environment.

For SIMPLIFYCOMPLETE, the environment may also contain other types of constraints, which allows for the discovery of more opportunities to apply rewrite rules. In addition, SIMPLIFYCOMPLETE uses a more aggressive technique to ensure that constraints that participate in the same rewrite rule can find one another. Because the order in which constraints are represented inside of an **R**-expr does not influence the **R**-expr's semantic interpretation (rewrite rule 15), this rewriting pass operates using two passes. The first pass scans the conjunctive part of the **R**-expr using GATHERENVIRONMENT to collect the relevant constraints into the environment $\mathcal{C}$. This pass does not perform any rewriting as it does not necessarily have a complete perspective on the **R**-expr. Once all of the conjunctive constraints are collected, a second pass using SIMPLIFYCOMPLETE can perform efficient *local* rewrites on the **R**-expr, while also consider the environment $\mathcal{C}$. This combination allows for higher-level constraints to be combined, similar to how built-in values are propagated. For example, two `lessthan` constraints can be combined as in rewrite rule 28 to infer a third `lessthan` constraint.

```
1:  function SIMPLIFYCOMPLETE(R, C)
2:      ▷ Rewrites which require a larger context are included in this routine.
3:      if R matches Q+S :
4:          C_Q ← COPY(C)
5:          C_S ← COPY(C)
6:          Q' ← SIMPLIFYCOMPLETEMAIN(Q, C_Q)
7:          S' ← SIMPLIFYCOMPLETEMAIN(S, C_S)
8:          C ∪= C_Q ∩ C_S                ▷ Note: C may now contain constraints as well as bindings to variables
9:          return (C_Q − C)*Q' + (C_S − C)*S'   ▷ Constraints not in C are added back to their sub-R-exprs
10:     else if R matches Q*S :
11:         Q' ← SIMPLIFYCOMPLETE(Q, C)   ▷ Similar to the conjunctive rule for SIMPLIFYFAST, but
        recursive with SIMPLIFYCOMPLETE
12:         if Q' matches 0 : return 0    ▷ The first conjunct has failed, so it can stop processing early.
        Implements rewrite rule 10
13:         S' ← SIMPLIFYCOMPLETE(S, C)
14:         if S' matches 0 : return 0                          ▷ Implements rewrite rule 10
15:         if Q' matches 1 : return S'                         ▷ Implements rewrite rule 9
16:         if S' matches 1 : return Q'                         ▷ Implements rewrite rule 9
17:     else if R matches lessthan(A, B) :                      ▷ Example of constraint-joining rules §3.4
18:         if lessthan(B, A) ∈ C :
19:             return 0          ▷ In this case, there are two constraints that are inconsistent, so it rewrites as 0.
20:         else if lessthan(B, C) ∈ C and lessthan(A, C) ∉ C :
21:             ▷ This example case implements the transitivity rule for lessthan (described in §3.4). Below, we add
                the new conjunct to the environment (optional) and the R-expr returned.
22:             C ∪= lessthan(A, C)
23:             return lessthan(A, B)*lessthan(A, C)
24:         else if lessthan(C, A) ∈ C and lessthan(C, B) ∉ C :
25:             ...                   ▷ Similar to the above for the other side of propagating with lessthan
26:         return R
27:     else if R matches (X=f(U_1, ..., U_n)) :                ▷ Handle unification of non-ground terms
28:         if (X=g(B_1, ..., B_m)) ∈ C and (g ≠ f or n ≠ m) :  ▷ Mismatched functors, rewrite rule 1
29:             return 0                                        ▷ These functors are incompatible
```

```
        if (X=f(V_1, ..., V_n)) ∈ C :                          ▷ Matched functor, rewrite rule 2
            return (U_1=V_1)*···*(A_n=B_n)                     ▷ Unify the nested variables
        if OccursCheck(X, U_1, ..., U_n) :                     ▷ Occurs to identify cycles like: X=f(Z), Z=f(X)
            return 0                                           ▷ for which there does not exist a solution, hence return 0, rewrite rule 5
        return R
    else
        ▷ Omitted: We do not show pattern-matching branches that are the same as SimplifyFast above. In the
          omitted branches, the recursive calls are made to SimplifyComplete instead of SimplifyFast.
        ...
function SimplifyCompleteMain(R, C)
    ▷ This routine is called on the top-level expression, and on each disjunct of a disjunctive sub-expression.
    ▷ First pass: traverse R, adding all conjunctive constraints to the environment
    C ∪= GatherEnvironment(R)
    ▷ Second pass: use the environment to simplify this branch
    R' ← SimplifyComplete(R, C)
    for Q+S ∈ GatherBranches(R') :                             ▷ Branches may rewrite further
        for B ∈ [Q, S] :                                       ▷ Loop over the branches of the disjunction
            R_B ← R'{(Q+S) ↦ B}                                ▷ Replace the disjunction with only one of its branches
            C_B ← Copy(C)
            R_B' ← SimplifyComplete(R_B, C_B)
            if R_B' matches 0 :
                R' ← R'{B ↦ 0}                                 ▷ B was fully eliminated
            else
                R' ← R'{B ↦ (C_B − C) * B}                     ▷ New constraints are added to this branch.
    return R'
function GatherEnvironment(R)
    ▷ Returns an environment containing all of the conjunctive expressions in R.
    if R matches Q+S : return ∅                                ▷ Do not recurse into + expressions
    if R matches Q*S : return GatherEnvironment(Q) ∪ GatherEnvironment(S)
    ▷ Recurse on subexpression of aggregation and projection, merge their environments
    C ← {R}
    for S ∈ SubExpressions(R) :
        C ∪= GatherEnvironment(S)
    return C
function GatherBranches(R)
    ▷ Returns a list of disjunctive expressions. For example, in the case that this is called on the R-expr
      (R0+R1)*(R2+R3), then it will return the list of R-exprs: [R0+R1, R2+R3]. This allows for SimplifyComplete
      to identify places in the R-expr where it can branch by selecting R0 or R1 for example.
    if R matches Q+S :
        return [R]                                             ▷ This only returns the conjunctive constraints, so it does not recurse on Q or S
    ▷ Recurse on subexpression of aggregation and projection, merge their branches
    b ← [ ]
    for S ∈ SubExpressions(R) :
        b.concatenate(GatherBranches(S))                       ▷ This recurses into conjunctive expressions, therefore
    combining all disjunctions that it finds into a single list
```

```
71:     return b
```