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

Matthew Francis-Landau, Tim Vieira, Jason Eisner
mfl@cs.jhu.edu
Johns Hopkins University

WRLA 2020 October 21
R-exprs
(Relational expressions)
Term Rewriting

R-exprs
(Relational expressions)
Machine Learning
Database
Deductive Databases

Dynamic Programming
Logic Programming
AI

Search

:\--dyna.

Compile

Term Rewriting

R-exprs
(Relational expressions)

Care about what not how something is computed
Term Rewriting + Queries

R-exprs (Relational expressions)

+ Queries

R-exprs (+ Query)

Search

Dynamic Programming

AI

Logic Programming

Machine Learning

Databases

Deductive

Database

Machine Learning

Databases

Deductive

Database

Machine Learning

Databases

Deductive

Database

Care about what not how something is computed
Term Rewriting

Machine Learning

Database

Deductive Databases

Dynamic Programming

Logic Programming

Search

\[ \Delta \setminus \nabla \] 

\text{:-dyna.}

Compile

R-exprs

(Relational expressions)

+ Queries

Results

(Hopefully) Useful Representation for User

Done

Care about \textbf{what not how} something is computed

Term Rewriting

Relational expressions (+ Query)

Done
Dyna vs. Prior Work
Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>‗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Infinite</td>
<td>‗</td>
<td>‗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregation</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Turing</td>
<td>‗</td>
<td>‗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constraints</td>
<td></td>
<td>‗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Supported by all. Naïve strategies terminate due to finite.
### Dyna vs. Prior Work

<table>
<thead>
<tr>
<th>Feature</th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Combining rules and “facts” to infer new “facts”
## Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Infinite relations</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

E.g. can we represent the set of all positive integers, or all prime numbers
## Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Infinite</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>relations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregation</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

SELECT `sum(column)` FROM x

Important for weighted programs
Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>✗</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Infinite relations</td>
<td>✗</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregation</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Turing complete</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Is this a full programming language
### Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Finite</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Deductive</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Infinite relations</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Aggregation</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Turing complete</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Can expressions like: \( X < Y \land \land Y < X \) be identified as impossible
## Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deductive</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Infinite relations</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregation</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Turing complete</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constraints</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Dyna vs. Prior Work

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>Datalog</th>
<th>Prolog</th>
<th>CLP</th>
<th>Dyna</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Finite</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Deductive</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Infinite relations</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Aggregation</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Turing complete</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**WANT ALL THE THINGS**

**Hard to mix**
Aggregation + Infinite

- $m(\{X : X \geq 5\}) = \infty$
- $\{X : X \geq 5\} = 5$
- $\frac{1}{2^t} = 2$
Aggregation + Infinite

Aggregators
• OR – Exists A True Branch
  • Used in Prolog (:-)
  • Can stop early if find true value

\[ m(\{X : X \geq 5\}) = \infty \]
\[ \{X : X \geq 5\} = 5 \]
\[ \frac{1}{2^t} = 2 \]
Aggregation + Infinite

Aggregators

• OR – Exists A True Branch
  • Used in Prolog (:-)
  • Can stop early if find true value

• AND – Not exist false branch

\[ m(\{X : X \geq 5\}) = \infty \]
\[ \{X : X \geq 5\} = 5 \]
\[ \frac{1}{2^t} = 2 \]
Aggregation + Infinite

Aggregators
• OR – Exists A True Branch
  • Used in Prolog ( :- )
  • Can stop early if find true value
• AND – Not exist false branch
• Sum/Product – exhaustive expansion of non-identity contributions

\[
m(\{X : X \geq 5\}) = \infty \\
\{X : X \geq 5\} = 5 \\
\frac{1}{2^t} = 2
\]
Aggregation + Infinite

Aggregators
• OR – Exists A True Branch
  • Used in Prolog (: -)
  • Can stop early if find true value
• AND – Not exist false branch
• Sum/Product – exhaustive expansion of non-identity contributions
• Max/Min – Structured Search problem or exhaustive search

\[ m(\{X : X \geq 5\}) = \infty \]
\[ \{X : X \geq 5\} = 5 \]
\[ \frac{1}{2^t} = 2 \]
Aggregation + Infinite

**Aggregators**
- OR – Exists A True Branch
  - Used in Prolog (\( : \neg \))
  - Can stop early if find true value
- AND – Not exist false branch
- Sum/Product – exhaustive expansion of non-identity contributions
- Max/Min – Structured Search problem or exhaustive search

**Infinite Relations**
- Infinite .....  
  - Can’t use a naïve enumerate strategy unless it stops early
  - \( m(\{X : X \geq 5\}) = \infty \)
  - \( \{X : X \geq 5\} = 5 \)
  - \( \frac{1}{2^l} = 2 \)
Aggregation + Infinite

Aggregators
• OR – Exists A True Branch
  • Used in Prolog (: –)
  • Can stop early if find true value
• AND – Not exist false branch
• Sum/Product – exhaustive expansion of non-identity contributions
• Max/Min – Structured Search problem or exhaustive search

Infinite Relations
• $\sum_{i=0}^{\infty} ii = 0 \sum_{i=0}^{\infty} i = 0 \sum_{i=0}^{\infty} 1 2 i 1 1 2 i 2 i \sum_{i=0}^{\infty} 1 2 i = 2$
• $X : X \geq 5 XX : XX \geq 5 X : X \geq 5) = 5$
• $m( X : X \geq 5 XX : XX \geq 5 X : X \geq 5 ) = \infty$
• $: XX \geq 5 X : X \geq 5$
• Infinite .....  
  • Can’t use a naïve enumerate strategy unless it stops early
  • Require special rules to understand sequences
  • $m( \{X : X \geq 5\}) = \infty$
  • $\{X : X \geq 5\} = 5$
• $\frac{1}{2^i} = 2$
Dyna = Logic Programming + Aggregation
Dyna = Logic Programming + Aggregation

\[ a(I) : \neg b(I), c(I). \]

- pointwise logical AND
Dyna = Logic Programming + Aggregation

\[ a(I) :- b(I), c(I). \]

- pointwise logical AND

\[ a(I) = b(I) \times c(I). \]

- pointwise multiplication
Dyna = Logic Programming + Aggregation

\[ a(I) \leftarrow b(I), c(I). \]

- pointwise logical AND

\[ a(I) = b(I) \ast c(I). \]

- pointwise multiplication

\[ a \leftarrow b(I) \ast c(I). \]

- dot product

\[ a = \sum_i b_i \ast c_i \]
Dyna = Logic Programming + Aggregation

\[ a(I) := b(I), c(I). \]
  • pointwise logical AND

\[ a(I) = b(I) * c(I). \]
  • pointwise multiplication

\[ a += b(I) * c(I). \]
  • dot product

\[ a = \sum_{i} b_i * c_i \]

\( I \) can range over any value, not just integers.
Dyna = Logic Programming + Aggregation

\[ a(I) :- b(I), c(I). \]
- pointwise logical AND

\[ a(I) = b(I) \cdot c(I). \]
- pointwise multiplication

\[ a += b(I) \cdot c(I). \]
- dot product

\[ a(I, K) += b(I, J) \cdot c(J, K). \]
- matrix multiplication; could be sparse
  - \( J \) is free on the right-hand side, so we sum over it

\[
\begin{align*}
(a &= \sum_i b_i \cdot c_i) \\
(a_{i,k} &= \sum_j b_{i,j} \cdot c_{j,k})
\end{align*}
\]
Dyna = Logic Programming + Aggregation

\[ a(I) :- b(I), c(I). \]
- pointwise logical AND

\[ a(I) = b(I) * c(I). \]
- pointwise multiplication

\[ a += b(I) * c(I). \]
- dot product

\[ a(I,K) += b(I,J) * c(J,K). \]
- matrix multiplication; could be sparse
  - \( J \) is free on the right-hand side, so we sum over it

\[ a = \sum_i b_i * c_i \]

\[ a_{i,k} = \sum_j b_{i,j} * c_{j,k} \]
Dyna = Logic Programming + Aggregation

\[ a(I) :- b(I), c(I). \]

- pointwise logical AND

\[ a(I) = b(I) * c(I). \]

- pointwise multiplication

\[ a += b(I) * c(I). \]

- dot product

\[ \begin{align*}
a(I, K) &= b(I, J) * c(J, K). \\
& \quad \text{matrix multiplication; could be sparse} \\
& \quad \text{J is free on the right-hand side, so we sum over it}
\end{align*} \]
Dyna = Logic Programming + Aggregation

\( a(I) :- b(I), c(I). \)
  * pointwise logical AND

\( a(I) = b(I) \times c(I). \)
  * pointwise multiplication

\( a += b(I) \times c(I). \)
  * dot product

\[
\begin{align*}
  a(I,K) & += b(I,J) \times c(J,K). \\
  a & = \sum_i b_i \times c_i \\
  a_{i,k} & = \sum_j b_{i,j} \times c_{j,k}
\end{align*}
\]
  * matrix multiplication; could be sparse
  * \( J \) is free on the right-hand side, so we sum over it

\( b(I,I) += 1. \quad b(I,J) += 0. \)
  * Infinite identity matrix
Example Program: Shortest path
Example Program: Shortest path

distance(Start, Y) \text{ min} = distance(Start, X) + \text{ edge}(X, Y). \\
distance(Start, Start) \text{ min} = 0.
Example Program: Shortest path

distance(\text{Start}, \ Y) \ \text{min}= \ distance(\text{Start}, \ X) \ + \ \text{edge}(X, \ Y).

\text{distance}(\text{Start, Start}) \ \text{min}= 0.

Variables not present in the head of an expression are aggregated over like with the dot product example.
Example Program: Shortest path

distance(Start, Y) min = distance(Start, X) + edge(X, Y).
distance(Start, Start) min = 0.

Here the “min=“ aggregator only keeps the minimal value that we have computed.
Example Program: Shortest path

distance(Start, Y) min= distance(Start, X) + edge(X, Y).
distance(Start, Start) min= 0.

edge("a", "b") = 10.
edge("b", "c") = 2.
edge("c", "d") = 7.
Example Program: Shortest path

\[
\begin{align*}
\text{distance}(\text{Start}, Y) & \min = \text{distance}(\text{Start}, X), \\
\text{distance}(\text{Start}, \text{Start}) & \min = 0.
\end{align*}
\]

\[
\begin{align*}
\text{edge}("\text{a}", "\text{b}") & = 10. \\
\text{edge}("\text{b}", "\text{c}") & = 2. \\
\text{edge}("\text{c}", "\text{d}") & = 7.
\end{align*}
\]

Dyna programs are equivalent to the set of values they define.
Example Program: Shortest path

distance(Start, Y) \min = \text{distance}(\text{Start}, X) + \text{edge}(X, Y).
\text{distance}(\text{Start}, \text{Start}) \min = 0.

Defined for all cases where both arguments are equal

<table>
<thead>
<tr>
<th>Start</th>
<th>Y</th>
<th>distance(Start, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3.1415</td>
<td>3.1415</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>Y</th>
<th>distance(Start, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;a&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;b&quot;</td>
<td>10</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;c&quot;</td>
<td>12</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>&quot;d&quot;</td>
<td>19</td>
</tr>
<tr>
<td>&quot;b&quot;</td>
<td>&quot;b&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;b&quot;</td>
<td>&quot;c&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;b&quot;</td>
<td>&quot;d&quot;</td>
<td>9</td>
</tr>
<tr>
<td>&quot;c&quot;</td>
<td>&quot;c&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;c&quot;</td>
<td>&quot;d&quot;</td>
<td>7</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>&quot;d&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
Shortest Path (cont.)

distance(S, S) = 0.
Shortest Path (cont.)

distance(S, S) = 0.

<table>
<thead>
<tr>
<th>S</th>
<th>Y</th>
<th>distance(S, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3.1415</td>
<td>3.1415</td>
<td>0</td>
</tr>
</tbody>
</table>
Shortest Path (cont.)

distance(S, S) = 0.

<table>
<thead>
<tr>
<th>S</th>
<th>Y</th>
<th>distance(S, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3.1415</td>
<td>3.1415</td>
<td>0</td>
</tr>
</tbody>
</table>

\{<Arg_1, Arg_2, Result>: Arg_1 = Arg_2 \text{ AND Result} = 0\}
Shortest Path (cont.)

distance(S, S) = 0.

\[
\{\langle \text{Arg}_1, \text{Arg}_2, \text{Result} \rangle: \text{Arg}_1 = \text{Arg}_2 \text{ AND Result} = 0\}
\]
Shortest Path (cont.)

distance(S, S) = 0.

<Tuple of Named Variables

<table>
<thead>
<tr>
<th>S</th>
<th>Y</th>
<th>distance(S, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3.1415</td>
<td>3.1415</td>
<td>0</td>
</tr>
</tbody>
</table>

\{⟨Arg_1, Arg_2, Result⟩: Arg_1 = Arg_2 \text{ AND Result} = 0\}
Shortest Path (cont.)

distance(S, S) = 0.

Tuple of Named Variables

Executable Code Defines the Rule

<table>
<thead>
<tr>
<th>S</th>
<th>Y</th>
<th>distance(S, Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>3.1415</td>
<td>3.1415</td>
<td>0</td>
</tr>
</tbody>
</table>
Shortest Path (cont.)

\[
distance(S, S) = 0.
\]

\[
\left\langle \text{Arg}_1, \text{Arg}_2, \text{Result} \right\rangle: \text{Arg}_1 = \text{Arg}_2 \ \text{AND} \ \text{Result} = 0
\]

Tuple of Named Variables

Executable Code Defines the Rule

\[
distance(S, Y) = distance(S, X) + \text{edge}(X, Y).
\]
Shortest Path (cont.)

\[\text{distance}(S, S) = 0.\]

\[\{(\text{Arg}_1, \text{Arg}_2, \text{Result}) : \text{Arg}_1 = \text{Arg}_2 \text{ AND Result} = 0\}\]

Tuple of Named Variables

Executable Code Defines the Rule

\[\text{distance}(S, Y) = \text{distance}(S, X) + \text{edge}(X, Y).\]

*Because of recursion, it can not be expressed using the set builder notation*
\[ \text{distance}(\text{Start}, \ Y) = \text{edge}(X, \ Y) + \text{distance}(\text{Start}, \ X). \]
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
Result = edge(Arg2, X) + distance(Arg1, X).

Normalize with standard names for all arguments
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
    Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X))

R-expr to Call function by name
distance(Start, Y) = edge(X, Y) + distance(Start, X).

distance(Arg1, Arg2) :-
    E = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X))
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
    Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X))
(D is distance(Arg1, X))

Recursive call to distance
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
   Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X))
(D is distance(Arg1, X))
 builtin_plus(Result, E, D)
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
  Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X)) ∩
(D is distance(Arg1, X)) ∩
builtin_plus(Result, E, D)

Intersect the bag by multiplying the multiplicities and joining these expressions using the same variable names
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
   Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X)) ∩
(D is distance(Arg1, X)) ∩
builtin_plus(Result, E, D)

Over the tuple ⟨Arg1, Arg2, Result, E, D, X⟩
distance(Start, Y) = edge(X, Y) + distance(Start, X).

Result is distance(Arg1, Arg2) :-
Result = edge(Arg2, X) + distance(Arg1, X).

(E is edge(Arg2, X)) \∩
(D is distance(Arg1, X)) \∩
builtin_plus(Result, E, D)

proj(E, proj(D, proj(X, )))

Now Over the tuple ⟨Arg1, Arg2, Result⟩

Project out all local variables
What about Aggregation?

distance(S, X) \text{min} = \text{edge}(X, Y) + \text{distance}(S, Y).

• Any semi-group: min, max, sum, product, logical OR, logical AND
What about Aggregation?

distance($S$, $X$)_{\text{min}} = \text{edge}(X, Y) + \text{distance}(S, Y).

- Any semi-group: min, max, sum, product, logical OR, logical AND

(Result = \text{min}($\text{MinInputVariable}$, $R$))
What about Aggregation?

distance($S$, $X$) \(\text{min=\;}\) edge($X$, $Y$) + distance($S$, $Y$).

• Any semi-group: min, max, sum, product, logical OR, logical AND

(Result = min(MinInputVariable, $R$))
What about Aggregation?

distance(S, X) = min\(\text{edge}(X, Y) + \text{distance}(S, Y)\).

- Any semi-group: min, max, sum, product, logical OR, logical AND

\((Result=\text{min}(\text{MinInputVariable}, R))\)
What about Aggregation?

distance(S, X)_{\text{min}} = \text{edge}(X, Y) + \text{distance}(S, Y).

• Any semi-group: min, max, sum, product, logical OR, logical AND

(Result = \text{min}(\text{MinInputVariable}, R))

Resulting value from aggregation

New intermediate variable introduced (Like project)

R-expr composed on previous slide
Shortest Path All Together Now

distance(S, S) min = 0.
distance(S, X) min = edge(X, Y) + distance(S, Y).
Shortest Path All Together Now

distance(S, S) min= 0.
distance(S, X) min= edge(X, Y) + distance(S, Y).

Result is distance(Arg1, Arg2) min= Arg1=Arg2, Result=0.
Result is distance(Arg1, Arg2) min= Result=edge(Arg2, Y) + distance(Arg1, Y).
Shortest Path All Together Now

\[
distance(S, S) \min = 0.
\]
\[
distance(S, X) \min = \text{edge}(X, Y) + distance(S, Y).
\]
Result is \(\text{distance}(	ext{Arg1}, \text{Arg2}) \min = \text{Arg1} = \text{Arg2}, \ \text{Result} = 0\).
Result is \(\text{distance}(	ext{Arg1}, \text{Arg2}) \min = \text{Result} = \text{edge}(	ext{Arg2}, Y) + \text{distance}(	ext{Arg1}, Y)\).

\((\text{Arg1} = \text{Arg2}) \cap (\text{MinInput} = 0)\)
Shortest Path All Together Now

\[ \text{distance}(S, S) \min = 0. \]
\[ \text{distance}(S, X) \min = \text{edge}(X, Y) + \text{distance}(S, Y). \]

Result is \( \text{distance}(\text{Arg1}, \text{Arg2}) \min = \text{Arg1}=\text{Arg2}, \text{Result}=0. \)
Result is \( \text{distance}(\text{Arg1}, \text{Arg2}) \min = \text{Result}=\text{edge}(\text{Arg2}, Y) + \text{distance}(\text{Arg1}, Y). \)

\[(\text{Arg1}=\text{Arg2}) \cap (\text{MinInput}=0)\]

\( \text{proj}(E, \text{proj}(D, \text{proj}(Y, (E \text{ is edge}(\text{Arg2}, Y)) \cap (D \text{ is distance}(\text{Arg1}, Y)) \cap \text{builtin}_\text{plus}(\text{MinInput}, E, D) )))) \)
Shortest Path All Together Now

distance(S, S) min= 0.
distance(S, X) min= edge(X, Y) + distance(S, Y).

Result is distance(Arg1, Arg2) min= Arg1=Arg2, Result=0.
Result is distance(Arg1, Arg2) min= Result=edge(Arg2, Y) + distance(Arg1, Y).

((Arg1=Arg2) \ \cap \ (\text{MinInput}=0)) \ \cup \ 
\text{proj}(E, \text{proj}(D, \text{proj}(Y, \ 
(E \ \text{is edge}(Arg2, Y)) \ \cap \ (D \ \text{is distance}(Arg1, Y)) \ \cap \ \text{builtin_plus}(\text{MinInput}, E, D))))
Shortest Path All Together Now

\[ \text{distance}(S, S) \min= 0. \]
\[ \text{distance}(S, X) \min= \text{edge}(X, Y) + \text{distance}(S, Y). \]

Result is \( \text{distance}(\text{Arg1}, \text{Arg2}) \min= \text{Arg1}=\text{Arg2}, \text{Result}=0. \)
Result is \( \text{distance}(\text{Arg1}, \text{Arg2}) \min= \text{Result}=\text{edge}(\text{Arg2}, Y) + \text{distance}(\text{Arg1}, Y). \)

\[
\begin{align*}
\text{Result} &= \min(\text{MinInput}, \\
& (\text{Arg1}=\text{Arg2}) \land (\text{MinInput}=0)) \lor \\
& \text{proj}(E, \text{proj}(D, \text{proj}(Y, \\
& (E \text{ is edge}(\text{Arg2}, Y)) \land (D \text{ is distance}(\text{Arg1}, Y)) \land \text{builtin_plus}(\text{MinInput}, E, D) ))
\end{align*}
\]

The complete distance rule as a R-expr.
Manipulating R-exprs via Rewrites
Manipulating R-exprs via Rewrites

- A series of *semantic preserving* rewrites which attempt to *simplify* the expression
  - Look for a sub-R-expr which can be rewritten to be simpler, do so!
Manipulating R-exprs via Rewrites

• A series of *semantic preserving* rewrites which attempt to *simplify* the expression
  • Look for a sub-R-expr which can be rewritten to be simpler, do so!
• Non-deterministic: Any order of rewrites is acceptable
  • Requires searching through the entire R-expr to identify what can be rewritten/run
Manipulating R-exprs via Rewrites

• A series of *semantic preserving* rewrites which attempt to *simplify* the expression
  • Look for a sub-R-expr which can be rewritten to be simpler, do so!

• Non-deterministic: Any order of rewrites is acceptable
  • Requires searching through the entire R-expr to identify what can be rewritten/run

• Fair rewrites: non-normal form sub-expression are eventually rewritten
  • Important in the case of recursive programs
Manipulating R-exprs via Rewrites

• A series of *semantic preserving* rewrites which attempt to *simplify* the expression
  • Look for a sub-R-expr which can be rewritten to be simpler, do so!
• Non-deterministic: Any order of rewrites is acceptable
  • Requires searching through the entire R-expr to identify what can be rewritten/run
• Fair rewrites: non-normal form sub-expression are eventually rewritten
  • Important in the case of recursive programs
• Core rewrites are presented in the paper
R-expr Rewrites—Built-ins
R-expr Rewrites—Built-ins

\[
\text{builtin}_\text{plus}(X,Y,Z) \equiv \{(X,Y,Z) : X + Y = Z\}
\]
R-expr Rewrites—Built-Ins

\[
\text{builtin\_plus}(X,Y,Z) \equiv \{(X,Y,Z): X + Y = Z\}
\]

\[
\text{builtin\_plus}(1,2,Z) \rightarrow (Z=3)
\]
R-expr Rewrites—Built-ins

\[ \text{builtin}_\text{plus}(X,Y,Z) \equiv \{ (X,Y,Z) : X + Y = Z \} \]

\[ \text{builtin}_\text{plus}(1,2,Z) \rightarrow (Z=3) \]

\[ \text{builtin}_\text{plus}(1,Y,Z) \]

No rewrites available for:

\[ 1 + Y = Z \]

\[ Y = 1, Z = 2 \]
\[ Y = 2, Z = 3 \]
\[ Y = 3, Z = 4 \]
\[ ... \]
R-expr Rewrites—Built-ins

 builtin_plus(1,2,Z) → (Z = 3)

 builtin_plus(1,Y,Z)

 builtin_plus(1,Y,Z) \equiv \{ Y = Z \}

 Propagate the assignment to Z

 (Z=3)*builtin_plus(1,Y,Z) \rightarrow (Z=3)*builtin_plus(1,Y,3)
R-expr Rewrites—Built-ins

```
builtin_plus(1, 2, Z) \rightarrow (Z = 3)
builtin_plus(1, Y, Z)
(Z = 3) * builtin_plus(1, Y, Z) \rightarrow (Z = 3) * builtin_plus(1, Y, 3)
(Z = 3) * builtin_plus(1, Y, 3) \rightarrow (Z = 3) * (Y = 2)
```

**Propagate the assignment to** Z

**Built-ins support multiple modes for computation**
R-expr Rewrites—Built-ins

\[ (X,Y,Z) \equiv \{ (X,Y,Z) : X + Y = Z \} \]

Maps to the multiplicity of being contained in the bag

\[
\begin{align*}
\text{builtin_plus}(1,2,Z) &\rightarrow (Z=3) \\
\text{builtin_plus}(1,Y,Z) &\rightarrow (Z=3) \times \text{builtin_plus}(1,Y,Z) \\
(Z=3) \times \text{builtin_plus}(1,Y,Z) &\rightarrow (Z=3) \times \text{builtin_plus}(1,Y,3) \\
(Z=3) \times \text{builtin_plus}(1,Y,3) &\rightarrow (Z=3) \times (Y=2) \\
\end{align*}
\]

* and + are over the bag’s multiplicity

\[
\begin{align*}
\text{builtin_plus}(1,2,3) &\rightarrow 1 \\
\text{builtin_plus}(1,2,4) &\rightarrow 0 \\
\end{align*}
\]

Check assignment is consistent
Rewriting Example: Shortest Path

Distance \text{ is } \text{distance}("a", "c")
Rewriting Example: Shortest Path

\[ \text{Distance} \text{ is distance}("a", "c") \]

\[
(\text{Result} = \text{min}(\text{MinInput}, \\
(\text{Arg1} = \text{Arg2})*(\text{MinInput} = 0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
(\text{E is edge}(\text{Arg2}, X))*(\text{D is} \\
\text{distance}(\text{Arg1}, X)\text{)*bultin_plus}(E, D, \text{MinInput}))))))
\]
Rewriting Example: Shortest Path

**Distance** is `distance("a", "c")`

\[
\text{Result} = \min(\text{MinInput}, \\
(\text{Arg1}=\text{Arg2}) \cdot (\text{MinInput}=0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
(E \text{ is edge}(\text{Arg2}, X)) \cdot (D \text{ is} \\
\text{distance}(\text{Arg1}, X) \cdot \text{bultin_plus}(E, D, \text{MinInput})))))
\]

\[
\text{Distance} = \min(\text{MinInput}, \\
("a"="c") \cdot (\text{MinInput}=0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
(E \text{ is edge}("c", X)) \cdot (D \text{ is distance}("a", X) \cdot \text{bultin_plus}(E, D, \text{MinInput}))))
\]
Rewriting Example: Shortest Path

Distance is distance("a", "c")

\[
\text{Result} = \min(\text{MinInput}, \\
(\text{Arg1} = \text{Arg2}) \times (\text{MinInput} = 0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
(\text{E is edge}(\text{Arg2}, X)) \times (D \text{ is distance}(\text{Arg1}, X) \times \text{bultin_plus}(E, D, \text{MinInput})))) \\
(\text{Distance} = \min(\text{MinInput}, \\
("a" = "c") \times (\text{MinInput} = 0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
(\text{E is edge}("c", X)) \times (D \text{ is distance}("a", X) \times \text{bultin_plus}(E, D, \text{MinInput}))))
\]

0 Variables not equal
("a" = "c") → 0 Variables not equal
Rewriting Example: Shortest Path

Distance is distance("a", "c")

Program

\[
\text{Result} = \min(\text{MinInput}, \\
\text{Arg1} = \text{Arg2} \times (\text{MinInput} = 0) + \\
\text{proj}(E, \text{proj}(D, \text{proj}(X, \\
\text{E is edge(\text{Arg2, X})} \times (D \text{ is} \\
\text{distance(\text{Arg1, X})} \times \text{bultin_plus}(E, D, \text{MinInput})))))
\]

Rewrites Rules

0 Multiplicative annihilation
0 Variables not equal
0 * R \rightarrow 0 Multiplicative annihilation
Rewriting Example: Shortest Path

Distance is distance("a", "c")

Program

(Redult=min(MinInput, (Arg1=Arg2)*(MinInput=0) + proj(E, proj(D, proj(X, (E is edge(Arg2, X))*(D is distance(Argr1,X)*bultin_plus(E,D,MinInput))))))

Rewrites Rules

R Additive identity
0 Multiplicative annihilation
0 Variables not equal
0 + R → R Additive identity
Rewriting Example: Shortest Path

Distance is distance("a", "c")

\[(\text{Result}=\min(\text{MinInput}, (\text{Arg1}=\text{Arg2}) \cdot (\text{MinInput}=0) + \text{proj}(E, \text{proj}(D, \text{proj}(X, (E \text{ is edge}(\text{Arg2}, X)) \cdot (D \text{ is distance}(\text{Argr1}, X) \cdot \text{builtin_plus}(E, D, \text{MinInput}))))))\]
Rewriting Example: Shortest Path

Distance is distance("a", "c")

(Result=min(MinInput, 
(Arg1=Arg2)*(MinInput=0) + 
proj(E, proj(D, proj(X, 
  (E is edge(Arg2, X)))*(D is 
  distance(Arg1,X)*bultin_plus(E,D,MinInput)))))

(Distance=min(MinInput, 
("a"="c")*(MinInput=0) + 
proj(E, proj(D, proj(X, 
  (E is edge("c", X)))*(D is distance("a",X)*bultin_plus(E,D,MinInput)))))

R Additive identity
R Additive identity
0 Multiplicative annihilation

0 + R = R
Additive identity

Rewrites Rules
\[
\text{Distance} = \min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \\
E \text{ is edge("c", X})) \times (D \text{ is distance("a", X)} \times \text{bultin_plus}(E, D, \text{MinInput})))))
\]
(Distance=\min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \text{E is edge("c", X))})*\text{D is distance("a",X)})*\text{builtin_plus}(E,D,\text{MinInput})))

(\text{Result is edge(\text{Arg1, Arg2}) :-}
\text{Arg1}="a")*(\text{Arg2}="b")*(\text{Result}=10) + 
\text{Arg1}="b")*(\text{Arg2}="c")*(\text{Result}=2) + 
\text{Arg1}="c")*(\text{Arg2}="d")*(\text{Result}=7)
(\text{Distance}=\min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \text{proj}(E,)\text{edge}("c", X)))))) * (D \text{ is distance}("a", X)*\text{bultin_plus}(E, D, \text{MinInput})))

(\text{Result is edge}(\text{Arg1}, \text{Arg2}) : -
(\text{Arg1}="a")*(\text{Arg2}="b")*(\text{Result}=10) +
(\text{Arg1}="b")*(\text{Arg2}="c")*(\text{Result}=2) +
(\text{Arg1}="c")*(\text{Arg2}="d")*(\text{Result}=7)

(\text{Distance}=\min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \text{proj}(E,)\text{("c"="a")*(X="b")*(E=10)}) +
\text{proj}(E,)\text{("c"="b")*(X="c")*(E=2)}) +
\text{proj}(E,)\text{("c"="c")*(X="d")*(E=7)})
*(D \text{ is distance}("a", X)*\text{bultin_plus}(E, D, \text{MinInput}))))
(Distance=min(MinInput, proj(E, proj(D, proj(X, 
  (E is edge("c", X))*(D is distance("a",X)*bultin_plus(E,D,MinInput)))))

(Result is edge(Arg1, Arg2)) :-
  (Arg1="a")(Arg2="b")(Result=10) +
  (Arg1="b")(Arg2="c")(Result=2) +
  (Arg1="c")(Arg2="d")(Result=7)

(Program
Distance=min(MinInput, proj(E, proj(D, proj(X, 
  ("c"="a")(X="b")(E=10)+
  ("c"="b")(X="c")(E=2)+
  ("c"="c")(X="d")(E=7))

*(D is distance("a",X)*bultin_plus(E,D,MinInput)))))

1
0
0
Equality checks
("c"="c") \rightarrow 1
\[(\text{Distance}=\min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, (E \text{ is edge}("c", X)))(D \text{ is distance("a",X)\ast\text{builtin_plus}(E,D,\text{MinInput}))))))

\begin{align*}
(\text{Result is edge}(\text{Arg1}, \text{Arg2})) & :-
\begin{align*}
(\text{Arg1}="a") & \ast (\text{Arg2}="b") \ast (\text{Result}=10) + \\
(\text{Arg1}="b") & \ast (\text{Arg2}="c") \ast (\text{Result}=2) + \\
(\text{Arg1}="c") & \ast (\text{Arg2}="d") \ast (\text{Result}=7)
\end{align*}
\end{align*}
\]
\[(\text{Distance} = \min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, E \text{ is edge}("c", X)))))(D \text{ is distance}("a", X)\ast\text{bultin_plus}(E, D, \text{MinInput})))\]

Program

\[(\text{Result is edge}(\text{Arg1}, \text{Arg2}) :-
(\text{Arg1}="a")\ast(\text{Arg2}="b")\ast(\text{Result}=10) +
(\text{Arg1}="b")\ast(\text{Arg2}="c")\ast(\text{Result}=2) +
(\text{Arg1}="c")\ast(\text{Arg2}="d")\ast(\text{Result}=7)\)

\[(\text{Distance} = \min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, (("c"="a")\ast(X="b")\ast(E=10) +
("c"="b")\ast(X="c")\ast(E=2) +
("c"="c")\ast(X="d")\ast(E=7))\ast(D \text{ is distance}("a", X)\ast\text{bultin_plus}(E, D, \text{MinInput}))))))\]

Multiplicative identity

1 * R \rightarrow R

Multiplicative identity
\[
(Distance = \min(MinInput, \ \text{proj}(E, \ \text{proj}(D, \ \text{proj}(X, \\
\quad (E \text{ is edge}("c", X))*(D \text{ is distance}("a", X)*\text{bultin_plus}(E, D, MinInput))))))
\]

(\text{Result is edge}(\text{Arg1, Arg2}) : - \\
\quad (\text{Arg1}="a")*(\text{Arg2}="b")*(\text{Result}=10) + \\
\quad (\text{Arg1}="b")*(\text{Arg2}="c")*(\text{Result}=2) + \\
\quad (\text{Arg1}="c")*(\text{Arg2}="d")*(\text{Result}=7)
\]

\[
(Distance = \min(MinInput, \ \text{proj}(E, \ \text{proj}(D, \ \text{proj}(X, \\
\quad (("c"="a")*(X="b")*(E=10) + \\
\quad ("c"="b")*(X="c")*(E=2) + \\
\quad ("c"="c")*(X="d")*(E=7)) \\
\quad *(D \text{ is distance}("a", X)*\text{bultin_plus}(E, D, MinInput))))))
\]

\[
\begin{align*}
R & \rightarrow \text{Multiplicative identity} \\
R & \rightarrow \text{Multiplicative identity} \\
1 & \\
0 & \\
1 & * R \rightarrow R \\
\end{align*}
\]

\[
(Distance = \min(MinInput, \ \text{proj}(E, \ \text{proj}(D, \ \text{proj}(X, \\
\quad ((X="d")*(E=7)) \\
\quad *(D \text{ is distance}("a", X)*\text{bultin_plus}(E, D, MinInput))))))
\]

\[
\begin{align*}
1 & \rightarrow R \\
1 & \rightarrow R \\
1 & \rightarrow R \\
\end{align*}
\]

14
\[
\text{Distance} = \min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \\
\text{\quad (E \text{ is edge}("c", X))}))(D \text{ is distance("a",X)}\ast\text{bultin_plus}(E,D,\text{MinInput}))))
\]

\[
\begin{align*}
(\text{Result is edge}(\text{Arg1}, \text{Arg2})) : &- \\
\quad (\text{Arg1}="a") \ast (\text{Arg2}="b") \ast (\text{Result} = 10) + \\
\quad (\text{Arg1}="b") \ast (\text{Arg2}="c") \ast (\text{Result} = 2) + \\
\quad (\text{Arg1}="c") \ast (\text{Arg2}="d") \ast (\text{Result} = 7)
\end{align*}
\]

Program

\[
(\text{Distance} = \min(\text{MinInput}, \text{proj}(E, \text{proj}(D, \text{proj}(X, \\
\text{\quad (("c"="a")\ast(X="b")\ast(E=10) + \\
\text{\quad ("c"="b")\ast(X="c")\ast(E=2) + \\
\text{\quad ("c"="c")\ast(X="d")\ast(E=7)}) \\
\text{\quad (D is distance("a",X)}\ast\text{bultin_plus}(E,D,\text{MinInput}))))))
\]

Propagate values
Rewrites for Aggregators
Rewrites for Aggregators

\[(\text{Result} = \min(\text{MinInput}, (\text{MinInput} = 789))) \rightarrow (\text{Result} = 789)\]

A final value has been determined. Assign it to the Result Variable.
Rewrites for Aggregators

\[(\text{Result}=\min(\text{MinInput}, (\text{MinInput}=789))) \rightarrow (\text{Result}=789)\]

\[(\text{Result}=\min(\text{MinInput}, R+S)) \rightarrow \text{builtin\_min}(MR, MS, \text{Result})\]*
\[ (MR=\min(\text{MinInput}, R))*(MS=\min(\text{MinInput}, S))\]

Two disjunctive R-exprs can be split and processed individually
Rewrites for Aggregators

\[(\text{Result}=\min(\text{MinInput}, (\text{MinInput}=789))) \rightarrow (\text{Result}=789)\]

\[(\text{Result}=\min(\text{MinInput}, R+S)) \rightarrow \text{builtin\_min}(MR, MS, \text{Result}) \]
\[\quad (MR=\min(\text{MinInput}, R)) \cdot (MS=\min(\text{MinInput}, S))\]

\[(\text{Result}=\min(\text{MinInput}, 0)) \rightarrow (\text{Result}=\text{id}entity) \equiv (\text{Result}=\infty)\]
Rewrites for Aggregators

(Result = \text{min}(\text{MinInput}, (\text{MinInput}=789))) \rightarrow (\text{Result}=789)

(Result = \text{min}(\text{MinInput}, R+S)) \rightarrow \text{builtin\_min}(MR, MS, \text{Result})*
\quad (MR=\text{min}(\text{MinInput}, R))*(MS=\text{min}(\text{MinInput}, S))

(Result = \text{min}(\text{MinInput}, 0)) \rightarrow (\text{Result}=identity) \equiv (\text{Result}=\infty)

\text{not\_identity}(identity) \rightarrow 0
\text{not\_identity}(V) \rightarrow 1 \quad \text{if ground}(V) \&\& V \neq identity

(Result = \text{min}(\text{MinInput}, 0))*\text{not\_identity}(\text{Result}) \rightarrow 0

More “traditional” for aggregation to map empty to empty
Ongoing and Future Work
Ongoing and Future Work

• Memoization and Mixed-chaining of computation
  • R-exprs serve as a basis for representing incomplete computations and can be run in a myriad of different execution orders
  • Extended version of this paper to (hopefully) appear soon
Ongoing and Future Work

• Memoization and Mixed-chaining of computation
  • R-exprs serve as a basis for representing incomplete computations and can be run in a myriad of different execution orders
  • Extended version of this paper to (hopefully) appear soon

• Exploring and learning different execution orders
  • R-exprs capture what needs to be computed while leaving the order and how open to the runtime to decide
  • Much like a database optimizer, but for full, long running programs
Ongoing and Future Work

• Memoization and Mixed-chaining of computation
  • R-exprs serve as a basis for representing incomplete computations and can be run in a myriad of different execution orders
  • Extended version of this paper to (hopefully) appear soon

• Exploring and learning different execution orders
  • R-exprs capture what needs to be computed while leaving the order and how open to the runtime to decide
  • Much like a database optimizer, but for full, long running programs

• Compilation and optimization of R-exprs

Thank you

Questions?

github.com/matthewfl/dyna-R
arxiv.org/abs/2010.10503
mfl@cs.jhu.edu