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

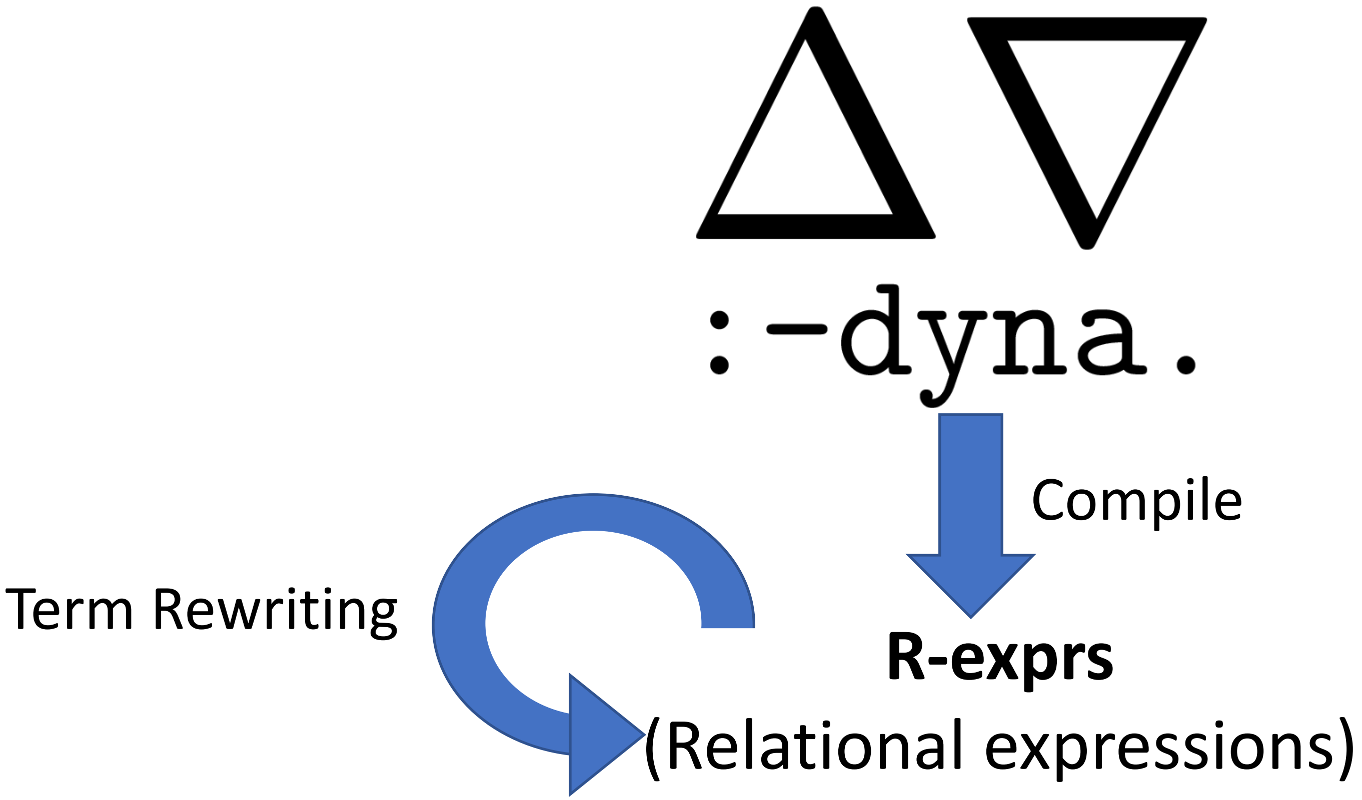
R-exprs

(Relational expressions)

Term Rewriting

R-exprs

(Relational expressions)


:-dyna.
Compile
Term Rewriting
R-exprs
(Relational expressions)

Search

Machine Learning

Dynamic Programming

Database

AI

Deductive Databases

:-dyna.

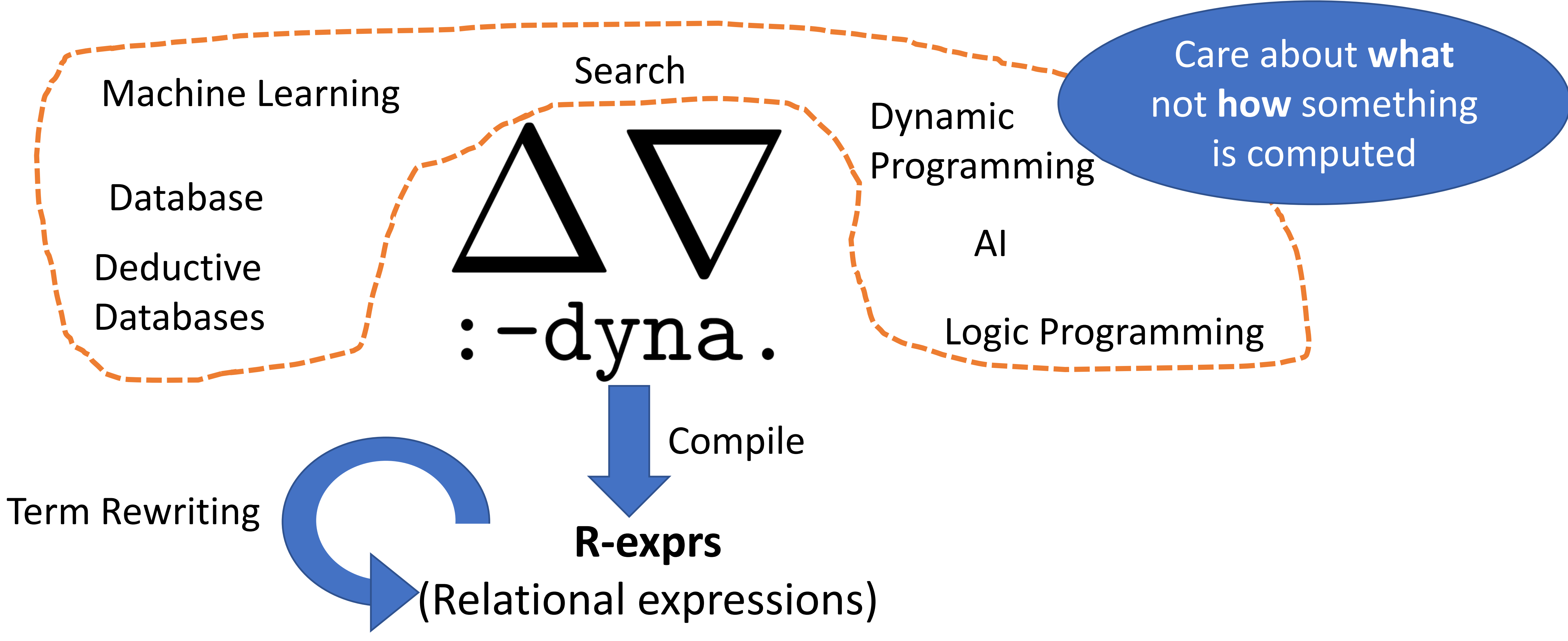
Logic Programming

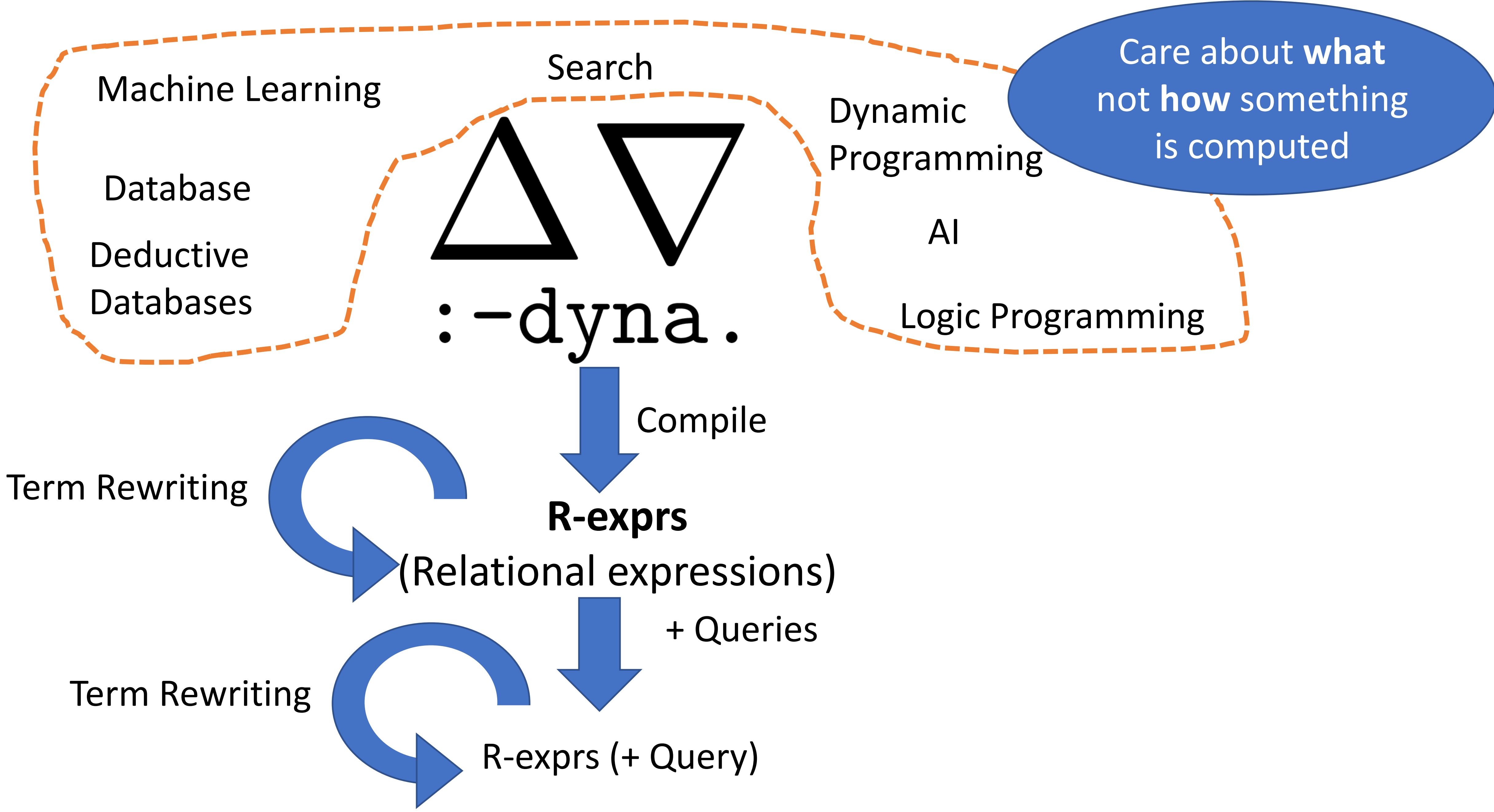
Compile

Term Rewriting

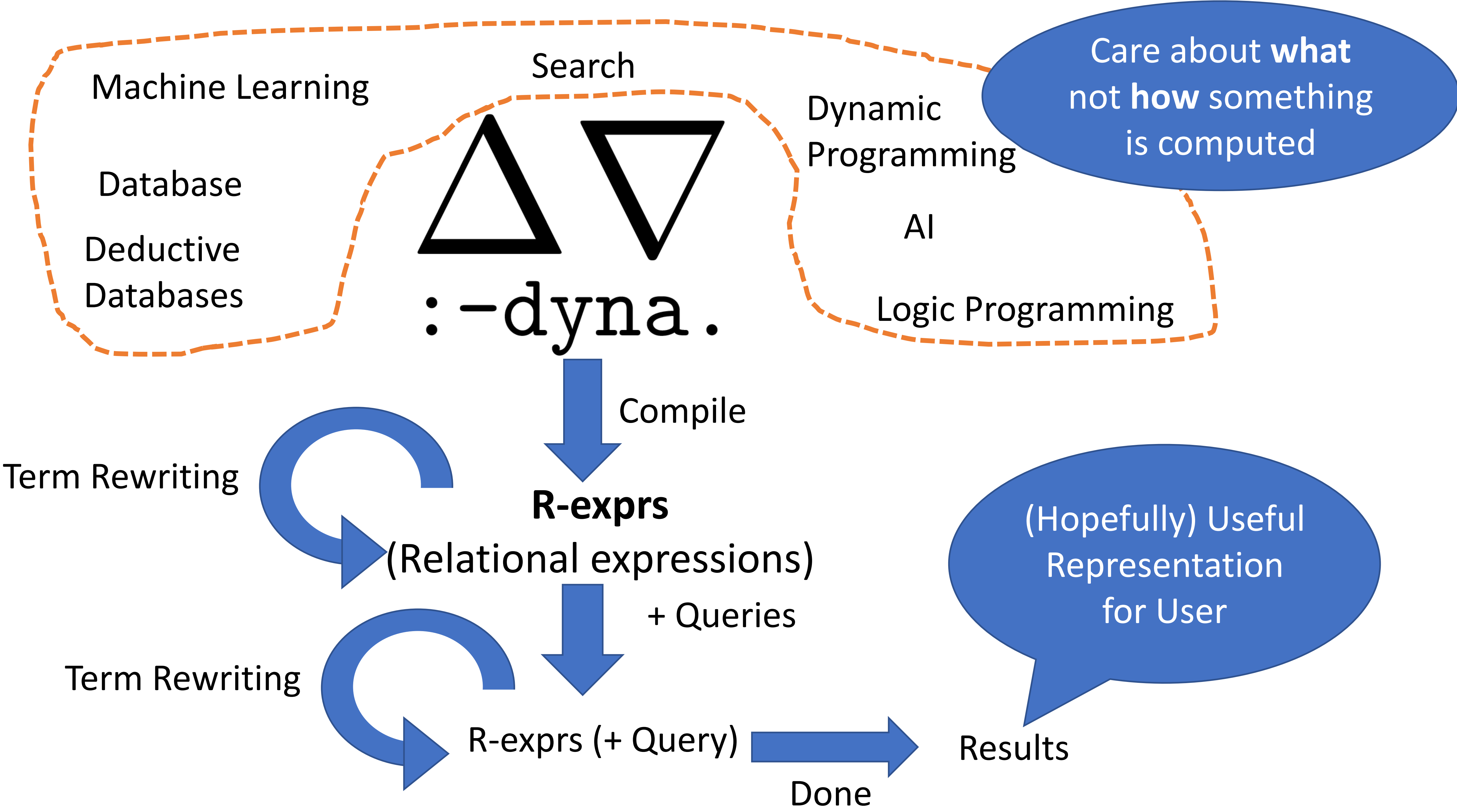
R-exprs

(Relational expressions)


Machine Learning
Search
Dynamic
Programming
Care about **what** not **how** something is computed
Database
Deductive
Databases
AI
Logic Programming
:-dyna.
Compile
Term Rewriting
R-exprs
(Relational expressions)



Machine Learning
Search
Database
Deductive Databases
Dynamic Programming
AI
Logic Programming
Care about **what** not **how** something is computed
:-dyna.
Compile
Term Rewriting
R-exprs
(Relational expressions)
+ Queries
Term Rewriting
R-exprs (+ Query)


Machine Learning
Search
Dynamic Programming
Care about **what** not **how** something is computed
Database
Deductive Databases
AI
Logic Programming
:-dyna.
Compile
Term Rewriting
R-exprs
(Relational expressions)
+ Queries
(Hopefully) Useful Representation for User
Term Rewriting
R-exprs (+ Query)
Results
Done

Dyna vs. Prior Work

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓

Supported by all. Naïve strategies terminate due to finite.

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓

Combining rules and "facts" to infer new "facts"

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓
Infinite relations	✗	✗	✓	✓	✓

E.g. can we represent the set of all positive integers, or all prime numbers

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓
Infinite relations	✗	✗	✓	✓	✓
Aggregation	✓	✓	✗	✗	✓

`SELECT sum(column) FROM x`

Important for weighted programs

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗				✓
Infinite relations	✗				✓
Aggregation	✓	✓	✗	✗	✓
Turing complete	✗	✗	✓	✓	✓

Is this a full programming language

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓
Infinite relations					✓
Aggregation					✓
Turing complete	✗			✓	✓
Constraints	✗	✗	✗	✓	✓

Can expressions like:
$X < Y \;\&\&\; Y < X$
be identified as
impossible

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓
Infinite relations	✗	✗	✓	✓	✓
Aggregation	✓	✓	✗	✗	✓
Turing complete	✗	✗	✓	✓	✓
Constraints	✗	✗	✗	✓	✓

Dyna vs. Prior Work

	SQL	Datalog	Prolog	CLP	Dyna
Finite	✓	✓	✓	✓	✓
Deductive	✗	✓	✓	✓	✓
Infinite relations	✗	✗	✓	✓	✓
Aggregation	✓	✓	✗	✗	✓
Turing complete	✗	✗	✓	✓	✓
Constraints	✗	✗	✗	✓	✓

WANT ALL THE THINGS

Hard to mix

Aggregation + Infinite

-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 2

Aggregation + Infinite

Aggregators

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

-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 2

Aggregation + Infinite

Aggregators

- OR – Exists A True Branch
  - Used in Prolog ($:-$)
  - Can stop early if find true value
- AND – Not exist false branch

-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 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

-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 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

-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 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

- Infinite .....
  - Can't use a naïve enumerate strategy unless it stops early
-
  - $\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 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

- ∑ i=0 ∞ ii=0 ∑ i=0 ∞ ∞ ∑ i=0 ∞ 1 2 i 1 1 2 i 2 i 2 2 i ii 2 i 1 2 i = 2
- X :X≥5 XX :XX≥5 X :X≥5)=5
- m(X :X≥5 XX :XX≥5 X :X≥5)=∞
- : XX≥5 X : X≥5
- *Infinite* …..
  - Can’t use a naïve enumerate strategy unless it stops early
  - Require special rules to understand sequences$\mathrm{m}(\{X : X \geq 5\}) = \infty$
  - $\{X : X \geq 5\}) = 5$
- $\frac{1}{2^i}$ = 2

Dyna = Logic Programming + Aggregation

Dyna = Logic Programming + Aggregation

```
a(I) :- b(I), c(I).
```

- pointwise logical AND

Dyna = Logic Programming + Aggregation

```
a(I) :- b(I), c(I).
```

- pointwise logical AND

```
a(I) = b(I) * c(I).
```

- pointwise multiplication

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

$$\left(a = \sum_i b_i * c_i\right)$$

Dyna = Logic Programm… …gation

```
a(I) :- b(I), c(I).
```

- pointwise logical AND

```
a(I) = b(I) * c(I).
```

- pointwise multiplicati…

```
a += b(I) * c(I).
```

- dot product

I can range over any value, not just integers

$$\left(a = \sum_i b_i * c_i\right)$$

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).
```

$$\left(a = \sum_i b_i * c_i \right)$$

- dot product

```
a(I,K) += b(I,J) * c(J,K).
```

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k} \right)$$

- matrix multiplication; could be sparse
- `J` is free on the right-hand side, so we sum over it

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

$$\left(a = \sum_i b_i * c_i\right)$$

```
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

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

Dyna = Logic Programming + Aggregation

```
a(I) :- b(I), c(I).
```

- pointwise logical AND

```
a(I) = b(I) * c(I).
```

- pointwise multiplication

$$\left(a = \sum_i b_i * c_i\right)$$

```
a += b(I) * c(I).
```

- dot product

```
a(I,K) += b(I,J) * c(J,K).
```

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

- matrix multiplication; could be sparse
- `J` is free on the right-hand side, so we sum over it

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

$$\left(a = \sum_i b_i * c_i\right)$$

```
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

$$\left(a_{i,k} = \sum_j b_{i,j} * c_{j,k}\right)$$

```
b(I,I) += 1.    b(I,J) += 0.
```

- *Infinite* identity matrix

Example Program: Shortest path

Example Program: Shortest path

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

Example Program: Shortest path

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

```
distance(Start, Y) min= distance(Start, X)
distance(Start, Start) min= 0.
```

Dyna programs are equivalent to the set of values they define

```
edge("a", "b") = 10.
edge("b", "c") = 2.
edge("c", "d") = 7.
```

Start	Y	distance(Start, Y)
"a"	"a"	0
"a"	"b"	10
"a"	"c"	12
"a"	"d"	19
"b"	"b"	0
"b"	"c"	2
"b"	"d"	9
"c"	"c"	0
"c"	"d"	7
"d"	"d"	0

Example Program: Shortest path

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

Defined for all cases where both arguments are equal

```
= 10.
) = 2.
d") = 7.
```

Start	Y	distance(Start, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

Start	Y	distance(Start, Y)
"a"	"a"	0
"a"	"b"	10
"a"	"c"	12
"a"	"d"	19
"b"	"b"	0
"b"	"c"	2
"b"	"d"	9
"c"	"c"	0
"c"	"d"	7
"d"	"d"	0

Shortest Path (cont.)

```
distance(S, S) = 0.
```

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\{\langle Arg_1, Arg_2, Result\rangle : Arg_1 = Arg_2 \textbf{ AND } Result = 0\}$$

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\{\langle Arg_1, Arg_2, Result\rangle : Arg_1 = Arg_2 \textbf{ AND } Result = 0\}$$

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\{\underbrace{\langle Arg_1, Arg_2, Result\rangle}_{\text{Tuple of Named Variables}}: Arg_1 = Arg_2 \textbf{ AND } Result = 0\}$$

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\{\underbrace{\langle Arg_1, Arg_2, Result\rangle}_{\text{Tuple of Named Variables}} : \underbrace{Arg_1 = Arg_2 \textbf{ AND } Result = 0}_{\text{Executable Code Defines the Rule}}\}$$

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\underbrace{\{\langle Arg_1, Arg_2, Result \rangle}_{\text{Tuple of Named Variables}} : \underbrace{Arg_1 = Arg_2 \textbf{ AND } Result = 0\}}_{\text{Executable Code Defines the Rule}}$$

```
distance(S, Y) = distance(S, X) + edge(X, Y).
```

Shortest Path (cont.)

```
distance(S, S) = 0.
```

S	Y	distance(S, Y)
"foo"	"foo"	0
7	7	0
3.1415	3.1415	0

$$\{\underbrace{\langle Arg_1, Arg_2, Result\rangle}_{\text{Tuple of Named Variables}} : \underbrace{Arg_1 = Arg_2 \textbf{ AND } Result = 0}_{\text{Executable Code Defines the Rule}}\}$$

```
distance(S, Y) = distance(S, X) + edge(X, Y).
```

Because of recursion, it can not be expressed using the set builder notation

```
distance(Start, Y) = edge(X, Y) + distance(Start, X).
```

```
distance(Start, Y) = edge(X, Y) + distance(Start, X).
```

Normalize with standard names for all arguments

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

R-expr to Call function by name

```
distance(Start, Y) = edge(X, Y) + distance(Start, X).
```

Intermediate results are mapped to variables

```
istance(Arg1, Arg2) :-
it = 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).
```

```
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)
```

Built-in represented in the R-expr

```
distance(Start, Y) = edge(X, Y) + distance(Start, X).
```

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

```
(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 $\langle \mathrm{Arg1}, \mathrm{Arg2}, \mathrm{Result}, E, D, X \rangle$

```
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 $\langle \text{Arg1}, \text{Arg2}, \text{Result} \rangle$

Project out all
local variables

What about Aggregation?

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

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

What about Aggregation?

```
distance(S, X) 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= edge(X, Y) + distance(S, Y).
```

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

```
(Result=min(MinInputVariable, R))
```

R-expr composed on previous slide

What about Aggregation?

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

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

```
(Result=min(MinInputVariable, R))
```

New intermediate variable introduced (Like project)

R-expr composed on previous slide

What about Aggregation?

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

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

```
(Result=min(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= 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) $\cap_{\star}$ (MinInput=0)

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) ⩀ (MinInput=0)

proj(E, proj(D, proj(Y,
(E is edge(Arg2, Y)) ⩀ (D is distance(Arg1, Y)) ⩀ builtin_plus(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) ∩* (MinInput=0)) ⊎

(proj(E, proj(D, proj(Y,
(E is edge(Arg2, Y)) ∩* (D is distance(Arg1, Y)) ∩* builtin_plus(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).
```

The complete distance rule as a R-expr

```
(Result=min(MinInput,
  ((Arg1=Arg2) ∩* (MinInput=0)) ⊎
(proj(E, proj(D, proj(Y,
 (E is edge(Arg2, Y)) ∩* (D is distance(Arg1, Y)) ∩* builtin_plus(MinInput, E, D)
 )))
)
                              ))
```

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

`builtin_plus(X,Y,Z)` $\equiv \{\langle X, Y, Z\rangle : X + Y = Z\}$

R-expr Rewrites—Built-ins

`builtin_plus(X,Y,Z)` $\equiv \{\langle X, Y, Z\rangle : X + Y = Z\}$

`builtin_plus(1,2,Z)` → `(Z=3)`

`builtin_plus`
runs and its result is
assigned **Z**

R-expr Rewrites—Built-ins

`builtin_plus(X,Y,Z)` $\equiv \{\langle X, Y, Z\rangle : X + Y = Z\}$

`builtin_plus(1,2,Z) → (Z=3)`

`builtin_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

Propagate the assignment to Z

builtin_plus(… $Y = Z\}$

builtin_plus(1,Z …

builtin_plus(1,Y,Z)

(Z=3)*builtin_plus(1,Y,Z)→(Z=3)*builtin_plus(1,Y,3)

R-expr Rewrites—Built-ins

Propagate the assignment to Z

builtin_plus(… $Y = Z\}$

builtin_plus(1,…

builtin_plus(1,Y,Z)

(Z=3)*builtin_plus(1,Y,Z)→(Z=3)*builtin_plus(1,Y,3)

(Z=3)*builtin_plus(1,Y,3)→(Z=3)*(Y=2)

Built-ins support multiple *modes* for computation

R-expr Rewrites—Built-ins

* and +
are over the
bag's
multiplicity

`(X,Y,Z)` $\equiv \{\langle X,Y,Z\rangle : X + Y = Z\}$

`s(1,2,Z) → (Z=3)`

`builti_plus(1,Y,Z)`

Maps to the
multiplicity of
being contained in
the bag

`(Z=3)*builtin_plus(1,Y,Z)→(Z=3)*bui`

`(Z=3)*builtin_plus(1,Y,3)→(Z=3)*`

```
builtin_plus(1,2,3) → 1
builtin_plus(1,2,4) → 0
```

Check
assignment
is
consistent

Rewriting Example: Shortest Path

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

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

```
(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))))
```

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

```
(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))))
```

Rewrites Rules

```
 0      Variables not equal
("a"="c") → 0
```

Variables not equal

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

```
(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))))
```

Rewrites Rules

```
 0          Multiplicative annihilat
 0      Variables not equal
0 * R → 0
```

Multiplicative annihilation

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

```
(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))))
```

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")
```

Program

```
(Result=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))))
```

```
(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))))
```

Rewrites Rules

```
R          Additive identity
R          Additive identity
0          Multiplicative annihilation
```

0 + R → R   Additive identity

0 + R → R   Additive identity

Rewriting Example: Shortest Path

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

Program

```
(Result=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))))
```

```
(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))))
```

Rewrites Rules

```
R          Additive identity
R          Additive identity
0          Multiplicative annihilation
```

```
0 + R → R  Additive identity
0 + R → R  Additive identity
```

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

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

Program

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

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

Program

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

```
(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))))
```

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

Program

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

```
(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") → 1
```

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

Program

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

```
(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))))
```

```
R          Multiplicative identity
1
0
0      Equality checks
```

1 * R → R Multiplicative identity

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

Program

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

```
(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))))
```

```
 R          Multiplicative identity
 R          Multiplicative identity
 1
 0
```

1 * R → R Multiplicative identity

1 * R → R Multiplicative identity

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

Program

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

```
(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))))
```

```
R          Multiplicative identity
R          Multiplicative identity
1
0
```

1 * R → R Multiplicative identity
1 * R → R Multiplicative identity

```
(Distance=min(MinInput, proj(E, proj(D, proj(X,
     ((X="d")*(E=7))
    *(D is distance("a",X)*bultin_plus(E,D,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))))
```

Program

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

```
(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))))
```

```
R          Multiplicative identity
R          Multiplicative identity
1
0
```

```
1 * R → R      Multiplicative identity
1 * R → R      Multiplicative identity
```

```
(Distance=min(MinInput, proj(E, proj(D, proj(X,
     ((X="d")*(E=7))
    *(D is distance("a",X)*bultin_plus(E,D,MinInput))))
```

Propagate values

```
(Distance=min(MinInput, proj(D,
     (D is distance("a","d")*bultin_plus(7,D,MinInput))))
```

Rewrites for Aggregators

Rewrites for Aggregators

```
(Result=min(MinInput, (MinInput=789))) → (Result=789)
```

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

Rewrites for Aggregators

```
(Result=min(MinInput, (MinInput=789))) → (Result=789)


(Result=min(MinInput, R+S)) → builtin_min(MR, MS, Result)*
                               (MR=min(MinInput, R))*(MS=min(MinInput, S))
```

Two disjunctive
R-exprs can be split
and processed
individually

Rewrites for Aggregators

```
(Result=min(MinInput, (MinInput=789))) → (Result=789)


(Result=min(MinInput, R+S)) → builtin_min(MR, MS, Result)*
                                (MR=min(MinInput, R))*(MS=min(MinInput, S))

(Result=min(MinInput, 0)) → (Result=identity) ≡ (Result=∞)
```

Rewrites for Aggregators

```
(Result=min(MinInput, (MinInput=789))) → (Result=789)


(Result=min(MinInput, R+S)) → builtin_min(MR, MS, Result)*
                               (MR=min(MinInput, R))*(MS=min(MinInput, S))


(Result=min(MinInput, 0)) → (Result=identity) ≡ (Result=∞)


not_identity(identity) →0
not_identity(V) → 1    if ground(V) && V != identity

(Result=min(MinInput, 0))*not_identity(Result) → 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
- github.com/matthewfl/dyna-R arxiv.org/abs/2010.10503

Thank you

Questions?

github.com/matthewfl/dyna-R

arxiv.org/abs/2010.10503

mfl@cs.jhu.edu