SlideRule: A Domain-Specific Language for Rewrite Rule Inference Using Equality Saturation

Anjali Pal, Brett Saiki, Oliver Flatt, Amy Zhu, Cynthia Richey, Ryan Tjoa, Max Willsey, Chandrakana Nandi, Zachary Tatlock

PLSE

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

SlideRule:
A Domain-Specific Language for Rewrite Rule Inference Using Equality Saturation

1. Equality Saturation
2. Rewrite Rule Inference
3. The SlideRule DSL

$(a * 2) / 2$ $\quad a$

$(x * y) / z = x * (y / z)$

$x * 2 = x << 1$

$x / x = 1$

$x * y = y * x$

$x * 1 = x$

$x = x * 1$

$(a * 2) / 2$

$a * (2 / 2)$

$a * 1$

a

$(a * 2) / 2$ $\quad$ a

$(x * y) / z = x * (y / z)$

$x * 2 = x \ll 1$

$x / x = 1$

$x * y = y * x$

$x * 1 = x$

$x = x * 1$

$(a * 2) / 2$

$((a * 1) * 2 / 2$

$((a * 1 * 1) * 2) / 2$

$(a * 2) / 2$

$(2 * a) / 2$

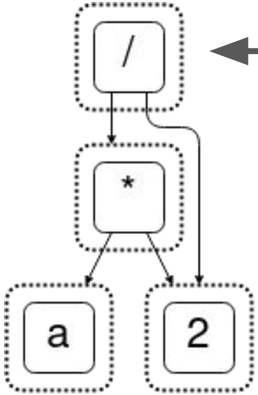
$(a * 2) / 2$

$(a * 2) / 2$

$(a \ll 1) / 2$

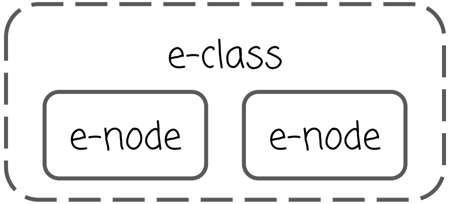
With Equality Saturation,
All of them, All the time

e-graphs

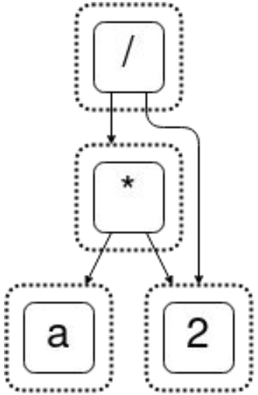


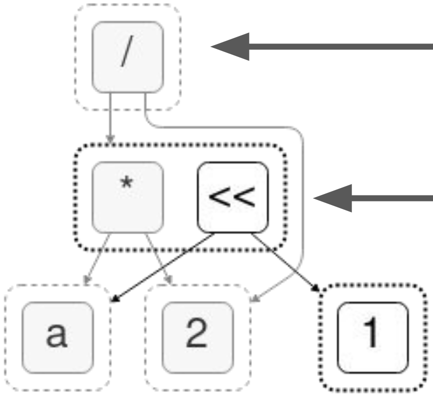
this e-class <u>represents</u>

$$(a * 2) / 2$$



growing an e-graph



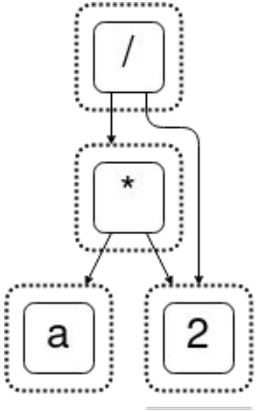


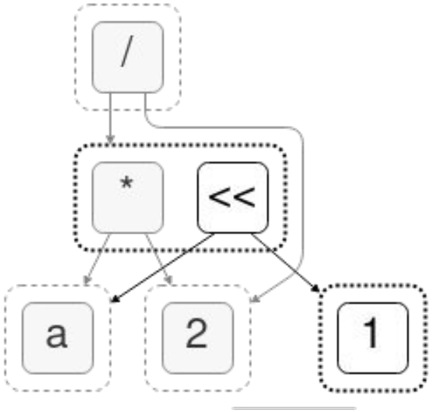
this e-class represents
(a * 2) / 2 **and** (a << 1) / 2

this e-class represents
(a * 2) **and** (a << 1)

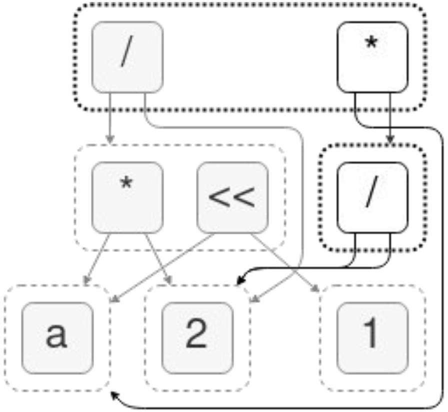
$x * 2 \rightarrow x \ll 1$

growing an e-graph





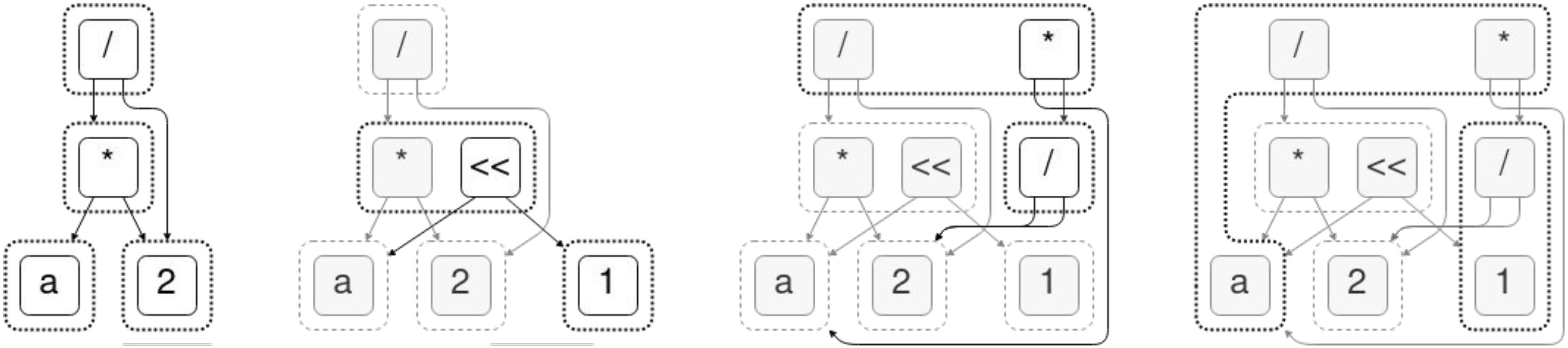
x * 2 → x << 1



(x * y) / z → x * (y / z)

growing an e-graph

a, a * 1,
a * 1 * 1, ...



x * 2 → x << 1

(x * y) / z → x * (y / z)

x / x → 1
x * 1 → x

Using e-graphs requires **high-quality** sets of rewrite rules, which are usually **written by hand** by **domain experts**, making **maintenance difficult**

1. Equality Saturation

2. Rewrite Rule Inference

3. The SlideRule DSL

Rule Inference

1. Term Enumeration
2. Candidate Generation
3. Rule Selection

Term Enumeration

EXPR :=

| var

| number

| (+ EXPR EXPR)

| (* EXPR EXPR)

a, b, 0, 1, (+ a a), (+ a b), (+ a 0), (+ a 1), (+ b a), (+ b b), (+ b 0), (+ b 1), (+ 0 a), (+ 0 b), (+ 0 0), (+ 0 1), (+ 1 a), (+ 1 b), (+ 1 0), (+ 1 1), (* a a), (* a b), (* a 0), (* a 1), (* b a), (* b b), (* b 0), (* b 1), (* 0 a), (* 0 b), (* 0 0), (* 0 1), (* 1 a), (* 1 b), (* 1 0), (* 1 1), (+ a (+ a a)), (+ a (+ a b)), (+ a (+ a 0)), (+ a (+ a 1)), (+ a (+ b a)), (+ a (+ b b)), (+ a (+ b 0)), (+ a (+ b 1)), (+ a (+ 0 a)), (+ a (+ 0 b)), (+ a (+ 0 0)), (+ a (+ 0 1)), (+ a (+ 1 a)), (+ a (+ 1 b)), (+ a (+ 1 0)), (+ a (+ 1 1)), (+ a (* a a)), (+ a (* a b)), (+ a (* a 0)), (+ a (* a 1)), (+ a (* b a)), (+ a (* b b)), (+ a (* b 0)), . . .

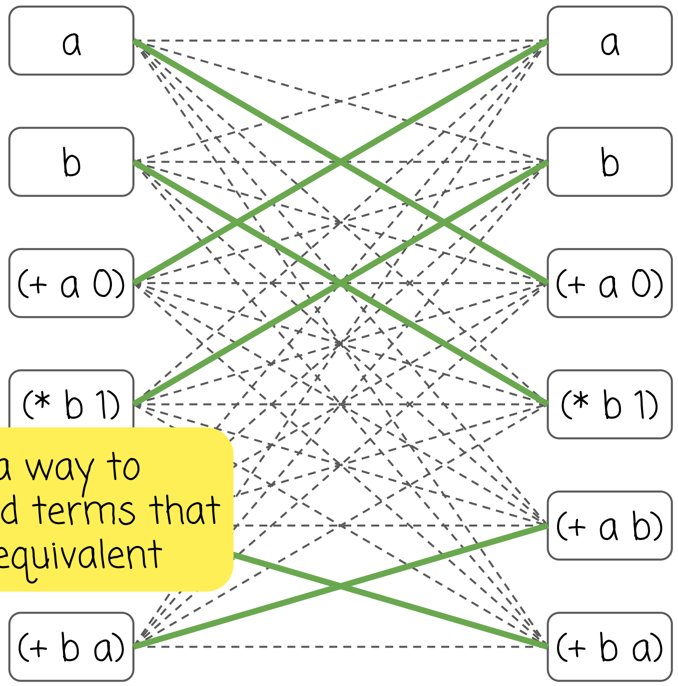
Candidate Generation

Naively, any pair of terms is a potential rewrite rule

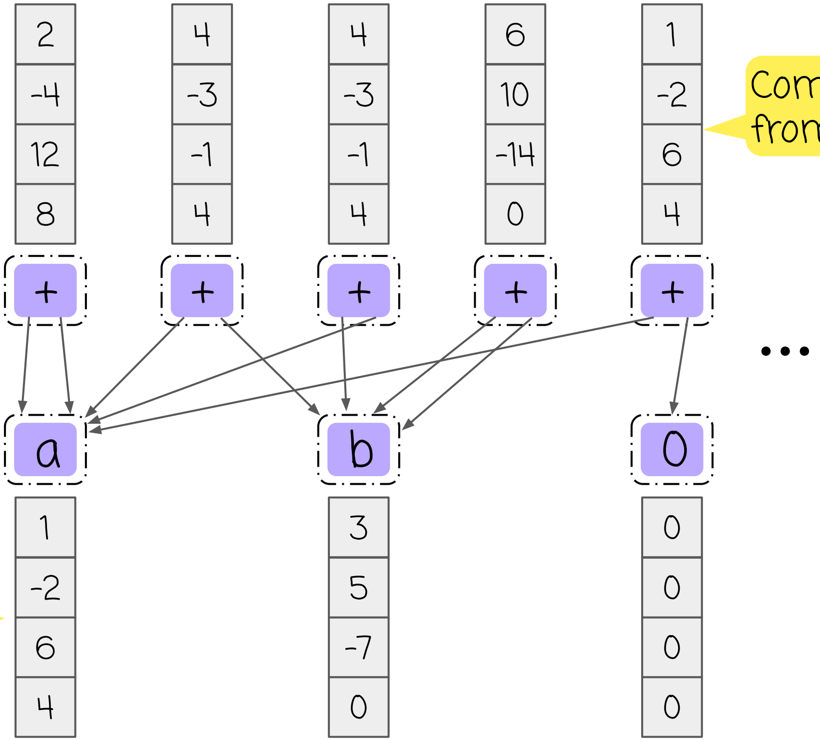
Even for only 6 terms, that leads to 36 potential rules

In this case only 6 of them are actually sound rewrites

We need a way to quickly find terms that are likely equivalent



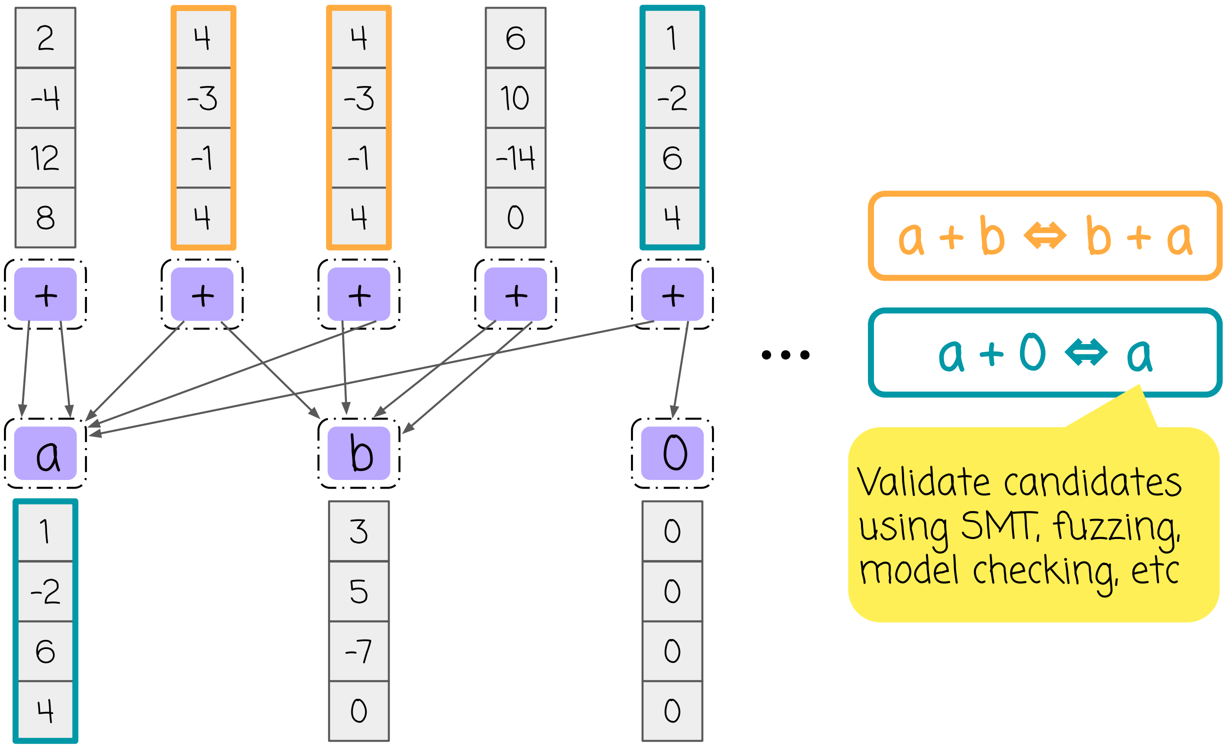
Candidate Generation



Compute cvecs from children

seed variables with concrete values from the domain

Candidate Generation



Rule Selection

$x + y \Leftrightarrow y + x$

$x + 0 \Leftrightarrow 0 + x$

$y + 0 \Leftrightarrow 0 + y$

$x * y \Leftrightarrow y * x$

$x * 1 \Leftrightarrow 1 * x$

$y * 1 \Leftrightarrow 1 * y$

$x + y \Leftrightarrow y + x$

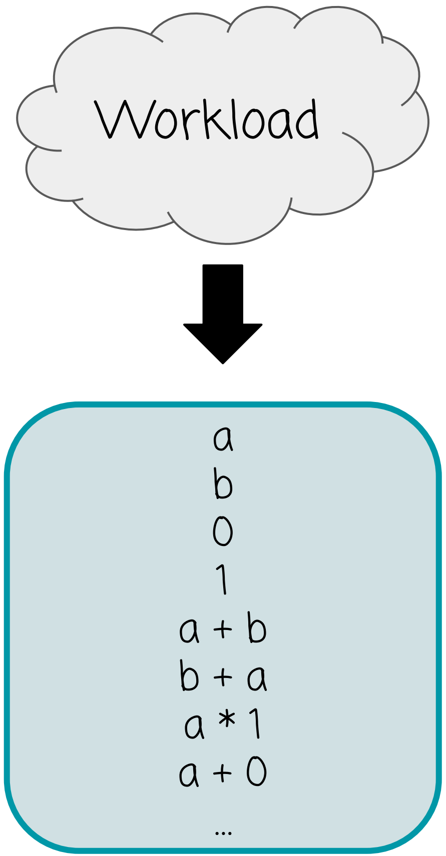
$x * y \Leftrightarrow y * x$

1. Equality Saturation

2. Rewrite Rule Inference

3. The SlideRule DSL

Workloads represent a set of terms



Plug

```
Plug takes two workloads, W1 and W2, and a
string, s

For each term in W1:
    For each occurrence of s in W1:
        For each term, t2, in W2:
            Make a term with t2 substituted for s
```

Plug

```
W1 = { X, foo(X), bar(X, X) }

W2 = { 1, 2, 3 }

Plug(W1, "X", W2)
```

```
1
2
3
```

```
foo(1)
foo(2)
foo(3)
```

```
bar(1, 1)
bar(1, 2)
bar(1, 3)
bar(2, 1)
bar(2, 2)
bar(2, 3)
bar(3, 1)
bar(3, 2)
bar(3, 3)
```

Workloads

```
G = { EXPR (~ EXPR) (+ EXPR EXPR) (* EXPR EXPR) }
leaves = { a b 0 1 }
d2 = Plug(G, "EXPR", leaves)
```

All terms up to depth 2

a	(+ b b)	(* a 0)
b	(+ b 0)	(* a 1)
0	(+ b 1)	(* b a)
1	(+ 0 a)	(* b b)
(~ a)	(+ 0 b)	(* b 0)
(~ b)	(+ 0 0)	(* b 1)
(~ 0)	(+ 0 1)	(* 0 a)
(~ 1)	(+ 1 a)	(* 0 b)
(+ a a)	(+ 1 b)	(* 0 0)
(+ a b)	(+ 1 0)	(* 0 1)
(+ a 0)	(+ 1 1)	(* 1 a)
(+ a 1)	(* a a)	(* 1 b)
(+ b a)	(* a b)	(* 1 0)
		(* 1 1)

Workloads

```
G = { EXPR (~ EXPR) (+ EXPR EXPR) (* EXPR EXPR) }
leaves = { a b 0 1 }
d2 = Plug(G, "EXPR", leaves)
```

`d3 = Plug(G, "EXPR", d2)`

All terms up to depth 3

Everything from d2, and:

```
(~ (~ a))
(~ (~ b))
...
(+ a (+ a a))
(+ a (+ a b))
(+ a (+ 1 1))
...
(+ (* a b) (* a a))
(+ (* a b) (* a b))
...
(* (* 1 1) (* 1 1))
```

Guided Search

```
base = { (OP VAL) }

ops = Plug(base, "OP", { sin cos tan })

all = Plug(ops, "VAL", { (/ PI 2) PI (* 2 PI) })

sound = Filter(all, !Contains("(tan (/ PI 2))"))
```

```
(sin (/ PI 2))   (cos (/ PI 2))   (tan PI)
(sin PI)         (cos PI)         (tan (* PI 2))
(sin (* PI 2))   (cos (* PI 2))
```

Filter out unsound terms

Guided Search

```
prods = Plug({ (* VAL VAL) }, "VAL", { a b 0 1 })

diff_of_prdcts =

   Plug({ (- VAL VAL) }, "VAL", prods)
```

Describe subsets of the term space

```
(* a a)  (* b a)
(* a b)  (* b b)
(* a 0)  (* b 0)
(* a 1)  (* b 1)
     ...
```

```
(- (* a a) (* a a))
(- (* a a) (* a b))
(- (* a a) (* a 0))
(- (* a a) (* a 1))
(- (* a b) (* a a))
        ...
```

Minimize

$x + y \Leftrightarrow y + x$

x

y

$x * y \Leftrightarrow y * x$

$x * 1 \Leftrightarrow 1 * x$

$y * 1 \Leftrightarrow 1 * y$

Rank sound candidates based on generality and select best one

$\boxed{x + y \Leftrightarrow y + x}$

$x * y \Leftrightarrow y * x$

$x + 0 \Leftrightarrow 0 + x$

$y + 0 \Leftrightarrow 0 + y$

$x * 1 \Leftrightarrow 1 * x$

$y * 1 \Leftrightarrow 1 * y$

Minimize

$\boxed{x + y \Leftrightarrow y + x}$

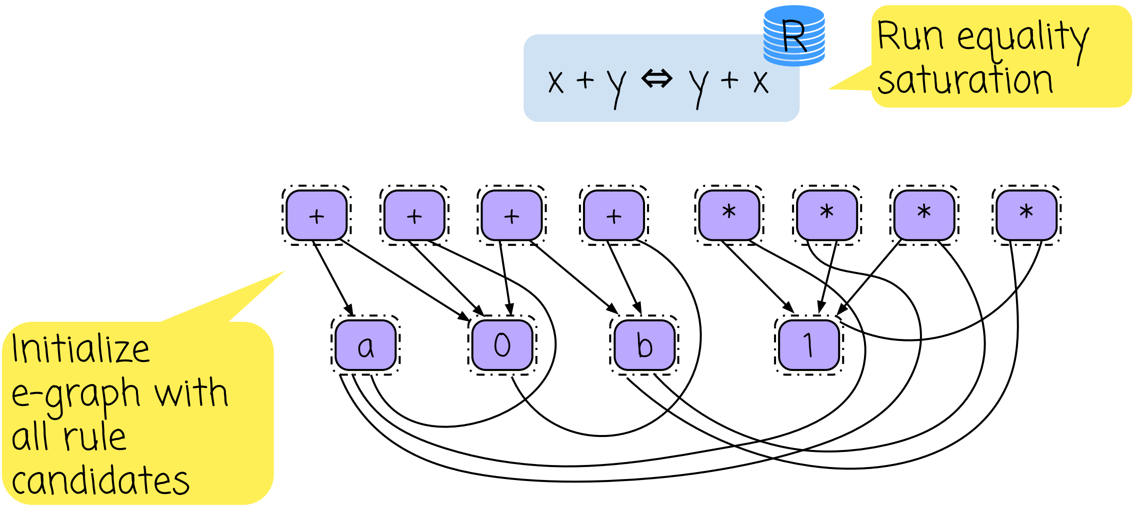
$x * y \Leftrightarrow y * x$

$x + 0 \Leftrightarrow 0 + x$

$y + 0 \Leftrightarrow 0 + y$

$x * 1 \Leftrightarrow 1 * x$

$y * 1 \Leftrightarrow 1 * y$



Minimize

x + y ⇔ y + x

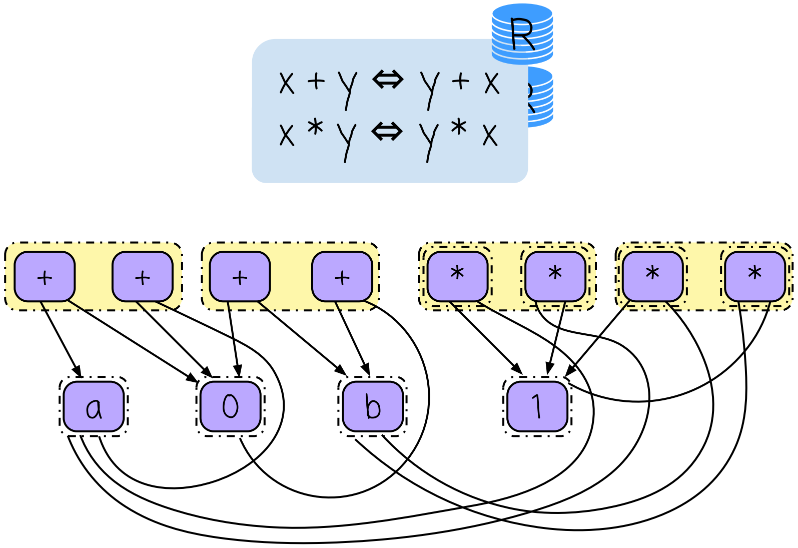
x * y ⇔ y * x

~~x + 0 ⇔ 0 + x~~

~~y + 0 ⇔ 0 + y~~

~~x * 1 ⇔ 1 * x~~

~~y * 1 ⇔ 1 * y~~



Rulesets

```
wkld = ...
```

Start with any Workload constructed using the operators shown previously

Rulesets

```
wkld = ...

egraph = wkld.to_egraph()
```

Initialize an e-graph that represents all the terms in the workload

Rulesets

```
wkld = ...

egraph = wkld.to_egraph()

candidates = egraph.find_candidates()
```

Find candidates using cvec matching

Rulesets

```
wkld = ...
egraph = wkld.to_egraph()
candidates = egraph.find_candidates()
(sound, unsound) =
   candidates.partition(|r| r.is_sound())
```

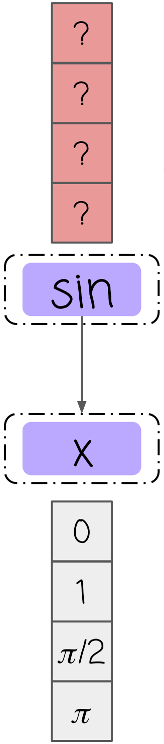
Find sound candidates using the domain-provided rule validator

Rulesets

```
wkld = ...

egraph = wkld.to_egraph()

candidates = egraph.find_candidates()

(sound, unsound) =
   candidates.partition(|r| r.is_sound())

rules = sound.minimize()
```

Minimize the sound candidates to select the best rules

Fast-Forwarding



Cvec matching is not possible in domains where equality is not decidable

Fast-Forwarding

- Run equality saturation in three phases, with different rules in each phase
- Strategically use copies of the e-graph to prevent adding too many new e-nodes and e-classes in each phase
- Learn rule candidates from merged e-classes

Think of these rules as "shortcuts"

Fast-Forwarding

Rules for rational arithmetic

a + b ⇔ b + a
a + (b + c) ⇔ (a + b) + c
a * 0 ⇔ 0
a * 1 ⇔ a
a * (b + c) ⇔ a * b + a * c
…

Rules expressing trig operators in terms of arithmetic

sin x ⇒ (cis x - cis -x) / 2i
cos x ⇒ (cis x + cis -x) / 2
tan x ⇒ sin x / cos x
…

Workload with trig terms

0	tan x	cos 0
1	sin PI	tan 0
PI	cos PI	sin PI/2
sin x	tan PI	cos PI/2
cos x	sin 0	tan PI/2

…

Fast-Forwarding

cos(PI/2 - x) <=> sin x

(1 - (cos(2x))) / 2 <=> $\sin^2 x$

(1 + (cos(2x))) / 2 <=> $\cos^2 x$

sin x * sin y <=> (cos(y - x) - cos(x + y)) / 2

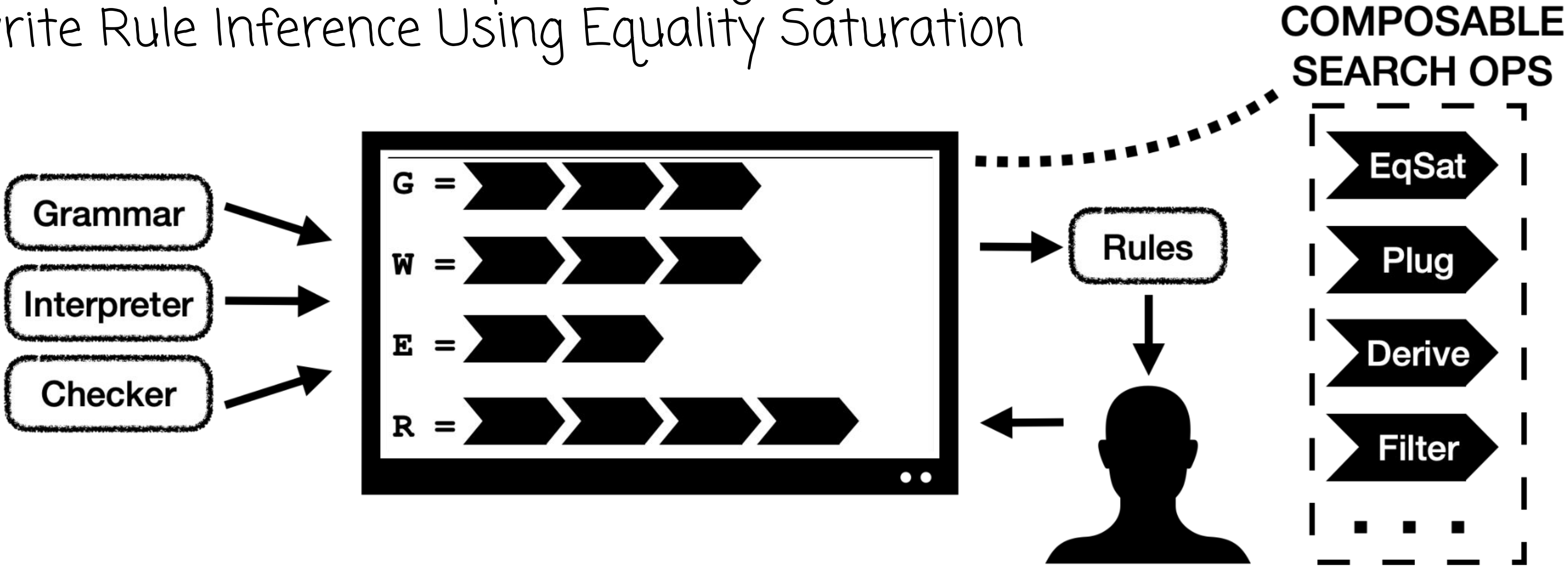
cos x * cos y <=> (cos(x + y) + cos(y - x)) / 2

(cos x * cos y) - (sin x * sin y) <=> cos(x + y)

(sin x * cos y) + (sin y * cos x) <=> sin(x + y)

We can learn all of these rules without evaluating a single trig expression https://github.com/uwplse/ruler

SlideRule: A Domain-Specific Language for Rewrite Rule Inference Using Equality Saturation











PLSE

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING