e-graphs, four ways

Max Willsey

2023

e-graphs, four ways

- e-graphs, as data structures
- e-graphs, as datalog
- ... but fast
- ... as datalog again

$$(a * 2) / 2$$

$$\Downarrow$$

$$a$$

$$(a * 2) / 2$$

$$=$$

$$a$$

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

rewrite it!

useful

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

$x / x = 1$

$x * 1 = x$

not so useful

$x * 2 = x << 1$

$x * y = y * x$

$x = x * 1$

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

happy path

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

$$x / x = 1$$

$$x * 1 = x$$

$(a * 2) / 2 \Rightarrow (a \ll 1) / 2$ ✖ wrong turn

loop

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

$a \Rightarrow a * 1 \Rightarrow a * 1 * 1 \Rightarrow \ldots$ infinite size

<u>pitfalls</u>

but <u>critical</u> for other inputs

$x * 2 = x \ll 1$

$x * y = y * x$

$x = x * 1$

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

which rewrite? when?

useful

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

$x / x = 1$

$x * 1 = x$

not so useful

$x * 2 = x \ll 1$

$x * y = y * x$

$x = x * 1$

$x / x = 1$

$x * 2 = x \ll 1$

$x = x * 1$

$x = 1 * x$

which rewrite? when?

<u>all of them! all the time!</u>

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

$x * y = y * x$

x / x = 1

x * 2 = x << 1

x = x * 1

x = 1 * x

e-graphs
+
equality saturation

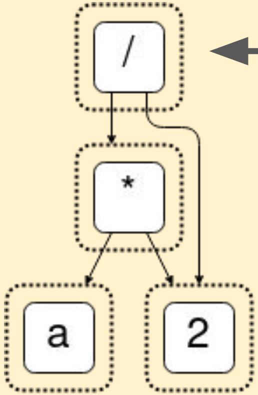
(z / y) * x = z / (y *

x * y = y * x

e-graphs, four ways

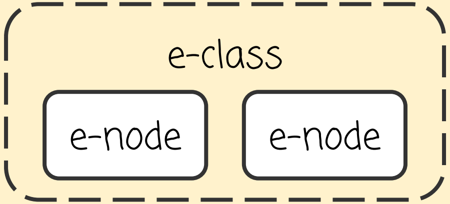
- <u>e-graphs, as data structures</u>
- e-graphs, as datalog
- … but fast
- … as datalog again

e-graphs?

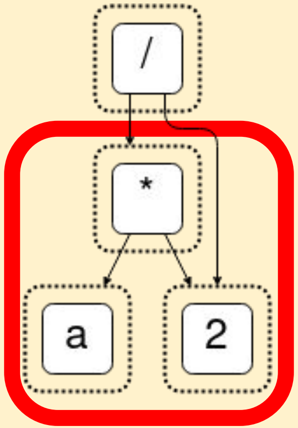


this e-class <u>represents</u>

(a * 2) / 2

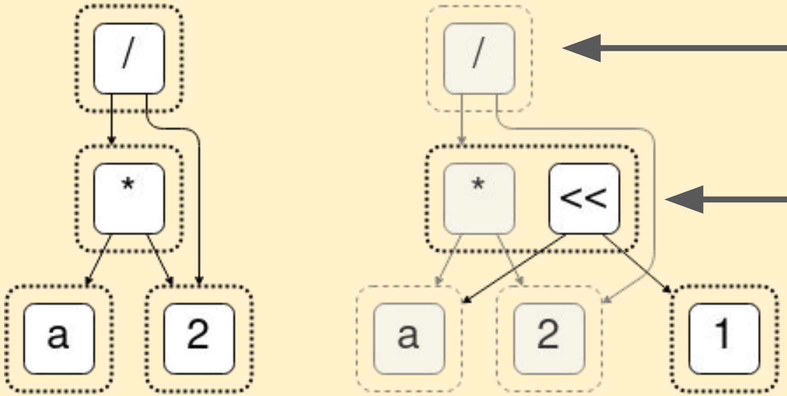


growing an e-graph



x * 2 → x << 1

growing an e-graph

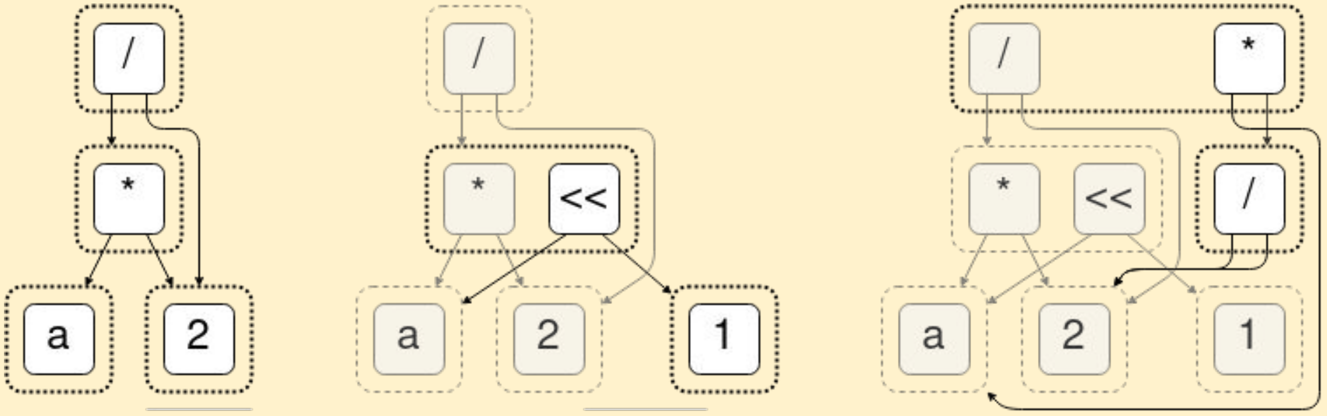


$x * 2 \rightarrow x \ll 1$

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

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

growing an e-graph

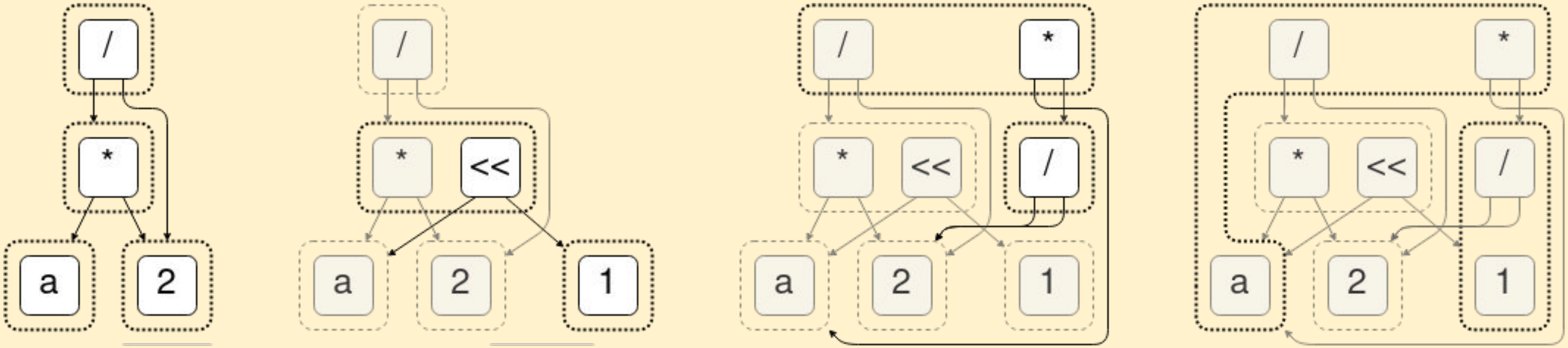


$x * 2 \rightarrow x \ll 1$

$(x * y) / z \rightarrow x * (y / z)$

e-graphs are compact

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



x * 2 → x << 1

(x * y) / z → x * (y / z)

x / x → 1
x * 1 → x

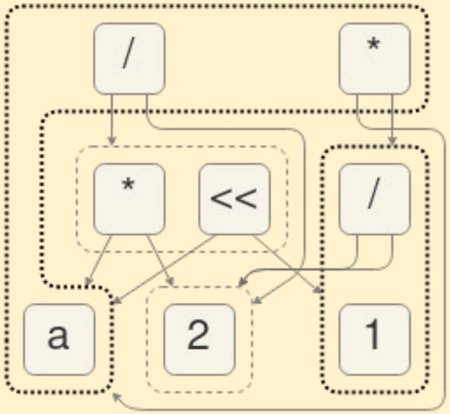
saturation

✓ $x * 2 \to x \ll 1$

✓ $(x * y) / z \to x * (y / z)$

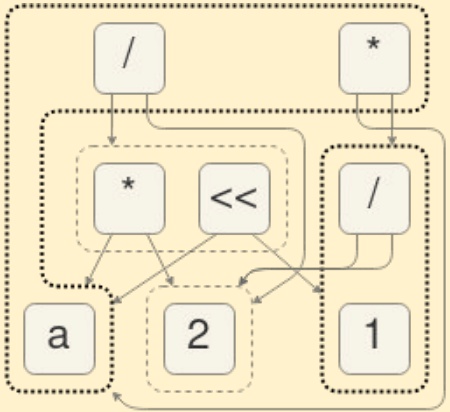
✓ $x / x \to 1$

✓ $x * 1 \to x$



extraction

this e-class represents
<u>(a * 2) / 2</u>, a, a * 1, ...



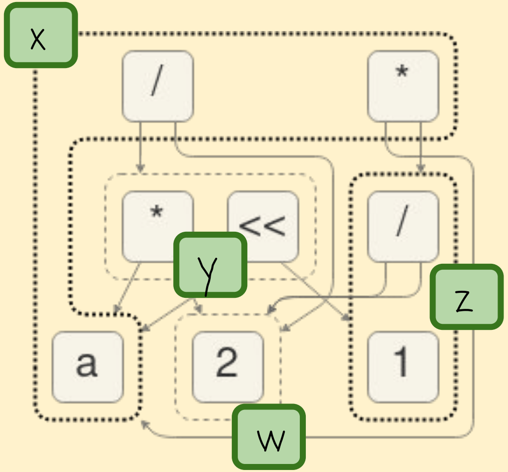
pick the smallest (cheapest) one

Knuth 76,
Generalization of Dijkstra's Algo.

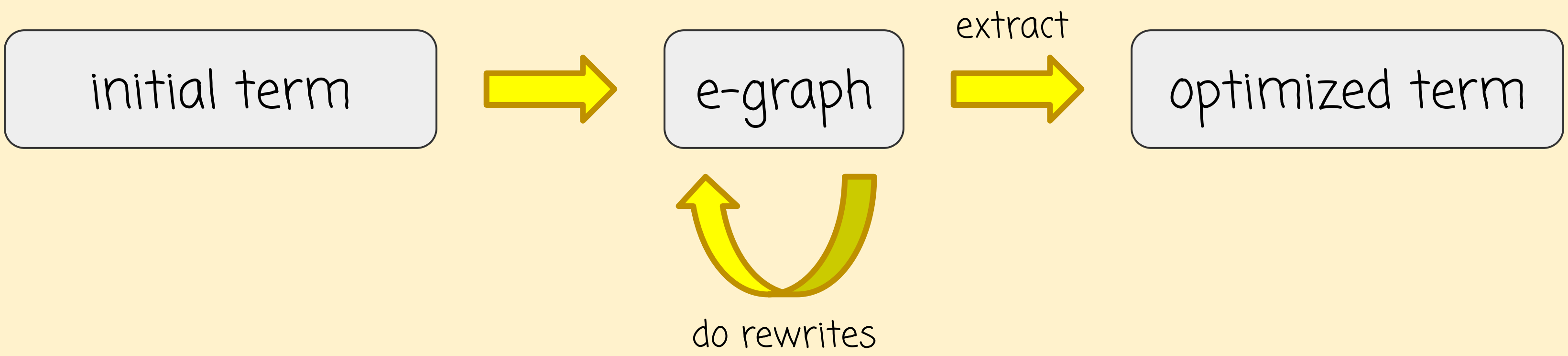
extraction

$$x = a + /(y, w) + *(x, z)$$
$$y = *(x, w) + \ll(x, z)$$
$$z = /(w, w) + 1$$
$$w = 2$$

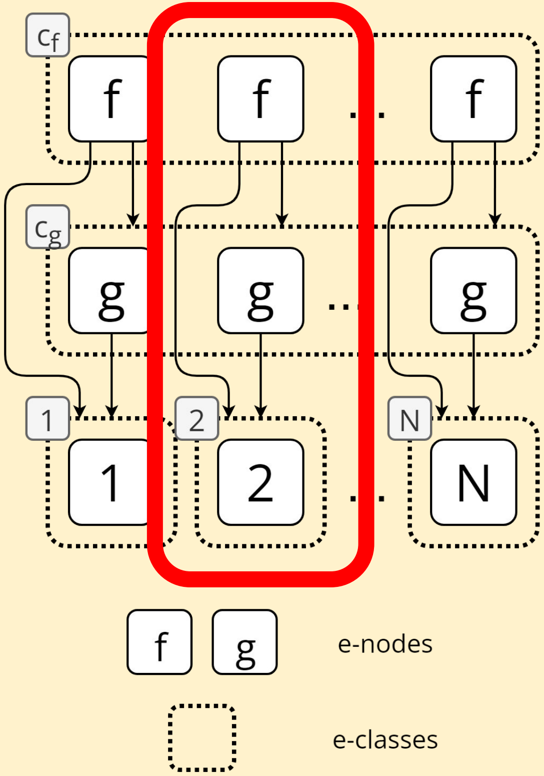
where f(a, b) =
min/+ semi-ring element
in terms of a, b



equality saturation

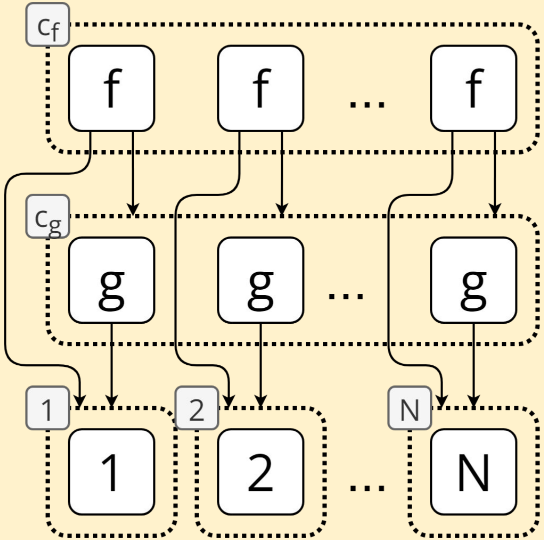


e-matching



pattern	substs	terms
$f(a, g(a))$	$\{a \mapsto 1\}$	$f(1, g(1))$
	$\{a \mapsto 2\}$	$f(2, g(2))$
	…	…
	$\{a \mapsto N\}$	$f(N, g(N))$

e-matching



pattern

f(a, g(a))

```
for e-class c in e-graph E:
  for f-node n₁ in c:
    subst = {root ↦ c, α ↦ n₁.child₁}
    for g-node n₂ in n₁.child₂:
      if subst[α] = n₂.child₁:
        yield subst
```

N^2 time, but only N matches

e-matching

- existing impls are **backtracking** based & **complex**
- doesn't help with **equality constraints**
- no **data complexity** results
    - NP-hard in pattern size... e-graph size??

more than rewriting

- there's more than *syntactic* rewriting
- sometimes, it's useful to consider *semantics*
  - $17 + 32 \rightarrow 49$, ...
- constant folding, nullability, tensor shape, non-zero, interval arithmetic, etc, ...

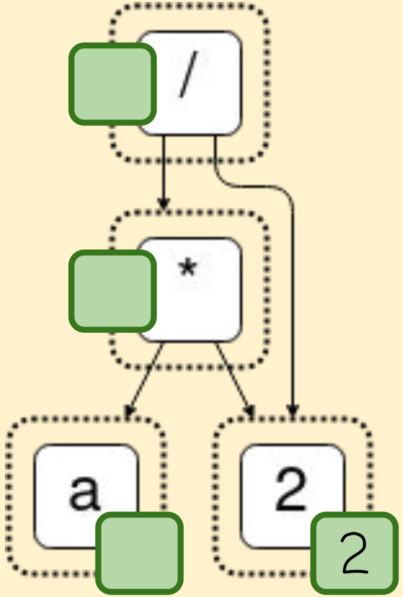
more than rewriting

analyses modulo equality

- uniform interface that works in many cases
- an understanding of analyses mean

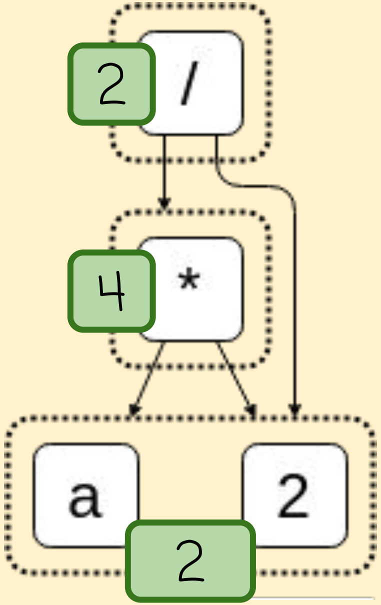
constant folding

- Option<Number> per eclass
- try to eval new e-nodes
- Option "or" on merge



constant folding

- Option<Number> per eclass
- try to eval new e-nodes
- Option "or" on merge
- it propagates up!



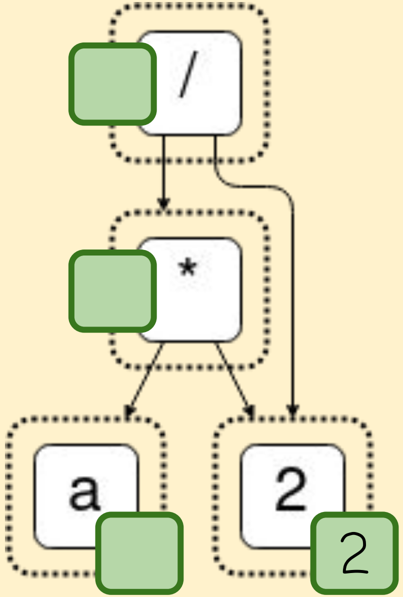
merge(a, 2)

e-class analysis

- 1 fact per e-class from a join-semilattice D
- make(n) → d_c
  - make a new analysis value for a new e-node
- join(d_{c1}, d_{c2}) → d_c
  - combine two analysis values
- modify(c) → c'
  - change the e-class (optionally)

constant folding

- D = Option<Number>
- make = eval
- join = option "or"
- modify = add the constant



detour: intervals

x in [0, 1]
y in [1, 2]

x + y in [1, 3]

detour: intervals

x in [0, 1]
y in [1, 2]

1 - 2y / (x + y) in [-3, 1/3]

= (x - y) / (x + y) in [-2, 0]

= 2x / (x + y) - 1 in [-1, 1]

intervals modulo equality

x in [0, 1]
y in [1, 2]

$$\left.\begin{aligned} & 1 - 2y / (x + y) && \text{in} && [-3, 1/3] \\ = {} & (x - y) / (x + y) && \text{in} && [-2, 0] \\ = {} & 2x / (x + y) - 1 && \text{in} && [-1, 1] \end{aligned}\right\} \mathbf{[-1, 0]}$$

e-class analysis uses

- lift program analyses to e-graphs
- conditional & dynamic rewrites
  - x / x = 1 iff x != 0
- can express other e-graph "hacks"
  - on-the-fly **extraction**

e-class analysis invariant

for each e-class

fixed point

$$\forall c \in G. \quad d_c = \bigvee_{n \in c} \mathrm{make}(n) \quad \text{and} \quad \mathrm{modify}(c) = c$$

Analysis data is LUB
(lattice properties)

egg: fast & easy e-graphs

- Rust library for generic e-graphs and eqsat
- packaged and documented: https://docs.rs/egg
- tutorials, industrial and academic users

POPL 21★

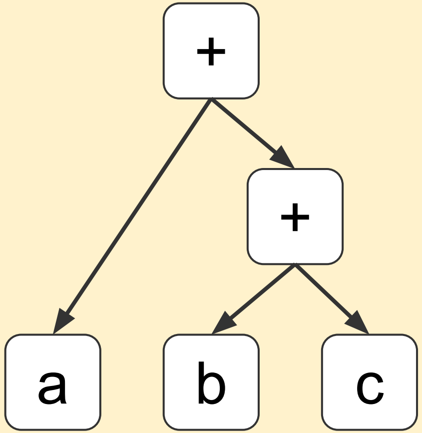
e-graphs, four ways

- e-graphs, as data structures
- <u>e-graphs, as datalog</u>
- … but fast
- … as datalog again

why?

- equality saturation is monotonic (-ish)
  - more equalities, more terms, no "destructive" rewrites
- e-matching is the bottleneck
  - it's a backtracking search for substitutions that satisfy a formula...

schema



"+" table

b	c	bc
a	bc	abc

rewrites as rules

- example: x + (y + z) -> (x + y) + z
- +(x, y, xy), +(xy, z, root2) <- +(x, yz, root), +(y, z, yz)
  - tempting to put root there
- not full existentials, just ADTs
  - existentials always "resolved" by FD, need a hashmap

what about the "e"?

- e-graph is an equivalence relation
  - congruence?
- pattern matching modulo equivalence
- equivalence is **<u>user-extensible</u>**!
  - think EGDs from chase

eq(x, y)

- just make an equivalence relation
  - symmetric, transitive, reflexive
- all joins modulo eq
  - R(x, y), R(y, z) becomes

    R(x, <u>y1</u>), eq(<u>y1</u>, <u>y2</u>), R(<u>y2</u>, z)

rewrites with eq

- example: x + (y + z) -> (x + y) + z
- +(x, y, xy), +(xy, z, root2), **eq(xy1, xy2)** <-
  +(x, yz1, root), +(y, z, yz2), **eq(yz1, yz2)**
- non-linear patterns tend to be cyclic
  - consider x + (y + x)

congruence

- eq(z1, z2) <-

  +(x1, y1, z1), +(x2, y2, z2), eq(x1, y1), eq(x2, y2)

doesn't work

- too slow
- various tricks don't fix it
    - specialized eqrel a la Souffle,
    - subsumption
    - see Yihong Zhang's thesis

lattices

- downside of e-class analyses: there's only one
- datalog has nice, cooperating "analyses"
  - mutually recursive rules
- requires recursive aggregation
  - LowerBound(expr, number)

e-graphs, four ways

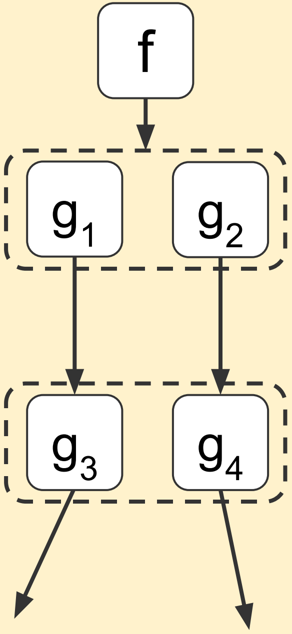
- e-graphs, as data structures
- e-graphs, as datalog
- **<u>… but fast</u>** PLDI 23
- … as datalog again

what was the problem?

- eq(x, y) too slow
  - n^2 size, etc.
- no canonicalization!

if you don't canonicalize...

- e-matching only yields canonical entries!
- f(g(g(x))
- $f(g_1(g_3(\ldots)))$, $f(g_2(g_3(\ldots)))$, $f(g_1(g_4(\ldots)))$, $f(g_2(g_4(\ldots)))$



canonicalize

- use a union-find to define leader(x)
  - leader(x) = leader(y) iff eq(x, y)
- e-graph: explicit maintenance
  - if x = y, **replace** x, y, with leader(x)
  - **collapse** e-nodes f(x), f(y) to f(leader(x))
  - massively shrinks the e-graph

canonicalize the db

- could use some form of subsumption
  - f(leader(x)) :- f(x)
  - f(x) <= f(y) :- x = leader(x), x != y
  - way too slow/hacky to implement in, e.g. Souffle
- let's just do what e-graphs do
  - congruence closure, "rebuilding"

no more eq

- eq relation/joins are gone!
- "semantic" equality becomes "syntactic" (again!)
- R(x, y1), R(y2, z), **eq(y1, y2)** becomes

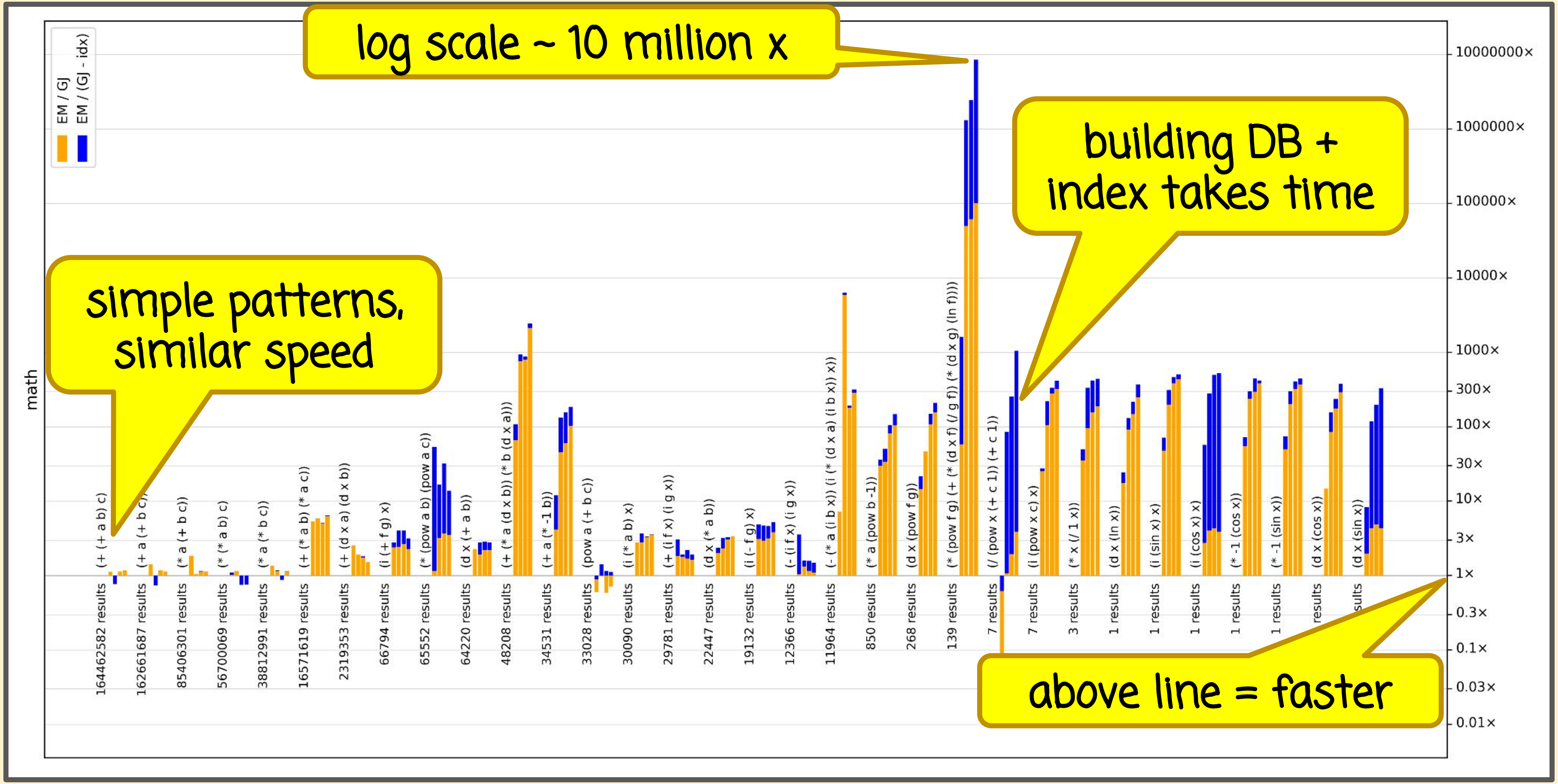
  R(x, y), R(y, z)

egglog

- datalog + functions + extensible equality
- examples
  - datalog: reachability
  - datalog + equality: reachability with node merging
  - eqsat: simple arithmetic optimization

egglog > eqsat

- simple implementation
  - separate optimization pass
- multipatterns
  - a x b = split(1, (a ++ c) x b)
  - a x c = split(2, (a ++ c) x b)
- incrementality via semi-naive



log scale ~ 10 million x
building DB + index takes time
simple patterns, similar speed
above line = faster
EM / GJ
EM / (GJ - idx)
math

functions + equality

- $f(a) = b$, $f(c) = d$
- equality is extensible! user asserts $a = c$
  - what happens?
  - what happens in datalog when $f(x, y)$ and $f(x, z)$
    - lattices / semirings

merge expressions

- (function foo (i64) i64 **:merge** (min old new))
- example
  - (set (foo 7) 5)
  - (set (foo 7) 4)
  - (foo 7) = 4

merge expressions

- also works for conflicts coming from equality
- ex: [interval arithmetic]
    - (function hi (Expr) Rational :merge (min old new))

      (function lo (Math) Rational :merge (max old new))

terms?

- (function mul (Expr Expr) Expr ...
  - :merge (union old new)
  - :default (mkset))
- congruence!
- labeled null? dynamic lattice?

the chase

- very similar to the Skolem chase
- EGDs capture FDs and "extensible" equality
- egglog has "stable" equality in a way
  - the UF prevents oscillation that can result in the chase

what's wrong?

- Built-in eq relation is really special
  - Requires infrastructure to do canonicalization
- semantic questions
  - why does this work? what's special about union-find?
- limits us to equality, and only one version of it

review

good

- fast queries
- fast congruence
- functions modulo eq
  - for e.g. e-class analysis

bad

- no terms!
  - only <u>families</u> of terms
- semantic questions
- only supports eq
  - rebuilding, extraction are special

e-graphs, four ways

- e-graphs, as data structures
- e-graphs, as datalog
- ... but fast
- <u>... as datalog again</u>

<u>WARNING</u>
braindump ahead

criteria

- no "built-in" equality
  - what are the features needed to support this?
- can't lose terms
- fast e-matching, congruence, etc.
  - via canonicalization?
- other relations
  - partial orders, **<u>indexed equality</u>**, etc.

new encoding

- can't canonicalize the term tables
- capture the good part of the union-find
- "term tables" Add, Mul are immutable
  - a "mutable" union-find captures the eqrel
  - canonicalization rules create new terms

new encoding

- (x + y) + z —> Add(x, y, xy), Add(xy, z, root)
- Add(x, y, xy1), Add(xy2, z, root),
- leader(xy1, xy2),
- leader(x, x), leader(y, y), leader(z, z)

egglog views

- views modulo "joining relations" like eq
- AddEq = { (leader(x), leader(y), leader(x))
  | (x, y, z) in Add }
- some kind of notion of monotonicity
  - why does this work? what's the algebraic structure?

monotonicity?

- some kind of notion of monotonicity
  - why does this work? what's the algebraic structure?
- leader is also aggregation over eqrel

payoff

- other relations than eq
  - non-symmetric: reduction relations
  - other equalities
  - Indexed equality: eq(expr, expr) -> semiring
    - context as a set of assumptions

example

- $a + 3b = a + b \pmod{3}$
- eq(x, y, mod)
- plus(a, b, **<u>ab</u>**), eq(ab, root, 3)   <-

  plus(a, m1, root), eq(m1, m2, T), mul(3, b, m2)
- eq(x, y, f), <- eq(x, y, k), factor(k, f)

payoff

- provenance
- semi-naive
  - can you re-derive congruence closure?
- top-down via demand transformation
- closer to "regular" datalog
  - other applications?

questions

- can any of this be implemented efficiently?
    - what algebraic structures are compatible with UF?
    - difficult IVM problem
    - queryable semirings?
- how to control the application of rules?
    - Demand transformation, explicit schedules, etc...
- a more flexible notion of monotonicity
- termination?