Relational Equality Saturation:
E-graphs meet Query Engines

Zachary Tatlock (UW Allen School) @ NWDS 2026-03-13


OFFLINE

Kernighan's Law:

- Debugging is twice as hard as writing the code in the first place.

Kernighan's Law:

- Debugging is twice as hard as writing the code in the first place.
- Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Kernighan's Law:

- Debugging is twice as hard as writing the code in the first place.
- Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Engineer complex systems
from **simple** and **modular** pieces!

PL Problems

1. Phase Ordering
2. Syntactic Brittleness
3. …

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

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

REWRITE!

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

REWRITE!

Useful

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

      x / x = 1

      x * 1 = x
```

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

REWRITE!

Useful

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

      x / x = 1

      x * 1 = x
```

Less Useful

```
x * 2 = x << 1

x * y = y * x

    x = x * 1
```

```
(a * 2) / 2
```

“happy path”

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

x / x = 1

x * 1 = x
```

```
(a * 2) / 2 ⇒ a * (2 / 2)
```

“happy path”

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

x / x = 1

x * 1 = x
```

```
(a * 2) / 2 ⇒ a * (2 / 2) ⇒ a * 1
```

“happy path”

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

x / x = 1

x * 1 = x
```

```
(a * 2) / 2 ⇒ a * (2 / 2) ⇒ a * 1 ⇒ a
```

“happy path”

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

x / x = 1

x * 1 = x
```

```
(a * 2) / 2 ⇒ a * (2 / 2) ⇒ a * 1 ⇒ a
```

“happy path”

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

      x / x = 1

      x * 1 = x
```

```
(a * 2) / 2
```

Pitfalls

```
x * 2 = x << 1
x * y = y * x
    x = x * 1
```

```
(a * 2) / 2 ⇒ (a << 1) / 2
```

Pitfalls

```
x * 2 = x << 1
x * y = y * x
    x = x * 1
```

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2`

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2`

Pitfalls

```
x * 2 = x << 1
x * y = y * x
    x = x * 1
```

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2`

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2` ❌

diverge

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2` ❌

diverge

a

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

(a * 2) / 2 ⇒ (a << 1) / 2 ❌ order

(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2 ❌

diverge

a ⇒ a * 1

Pitfalls

x * 2 = x << 1

x * y = y * x

x = x * 1

(a * 2) / 2 ⇒ (a << 1) / 2 ❌ order

(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2 ❌

diverge

a ⇒ a * 1 ⇒ a * 1 * 1

Pitfalls

x * 2 = x << 1

x * y = y * x

x = x * 1

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2` ❌

diverge

`a ⇒ a * 1 ⇒ a * 1 * 1 ⇒ …` ❌ infinite size

Pitfalls

`x * 2 = x << 1`

`x * y = y * x`

`x = x * 1`

`(a * 2) / 2 ⇒ (a << 1) / 2` ❌ order

`(a * 2) / 2 ⇒ (2 * a) / 2 ⇒ (a * 2) / 2` ❌

diverge

`a ⇒ a * 1 ⇒ a * 1 * 1 ⇒ …` ❌ infinite size

Critical for other inputs!

Pitfalls

```
x * 2 = x << 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
```

Less Useful

```
x * 2 = x << 1

x * y = y * x

    x = x * 1
```

PL Problems

1. Phase Ordering 😭
2. Syntactic Brittleness
3. …

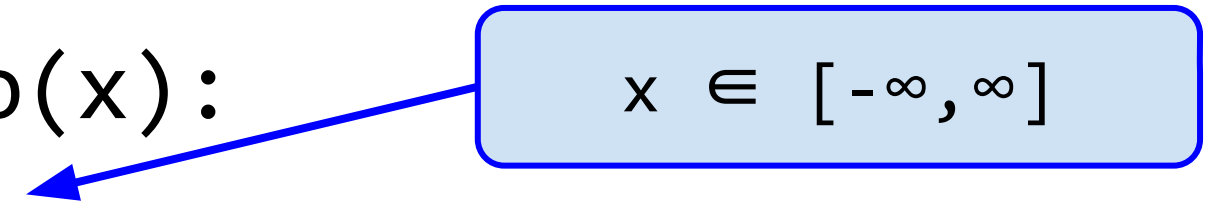
```
def foo(x):
    y = abs(x)
    return sqrt(y)
```

```
def foo(x):
    y = abs(x)
    return sqrt(y)
```

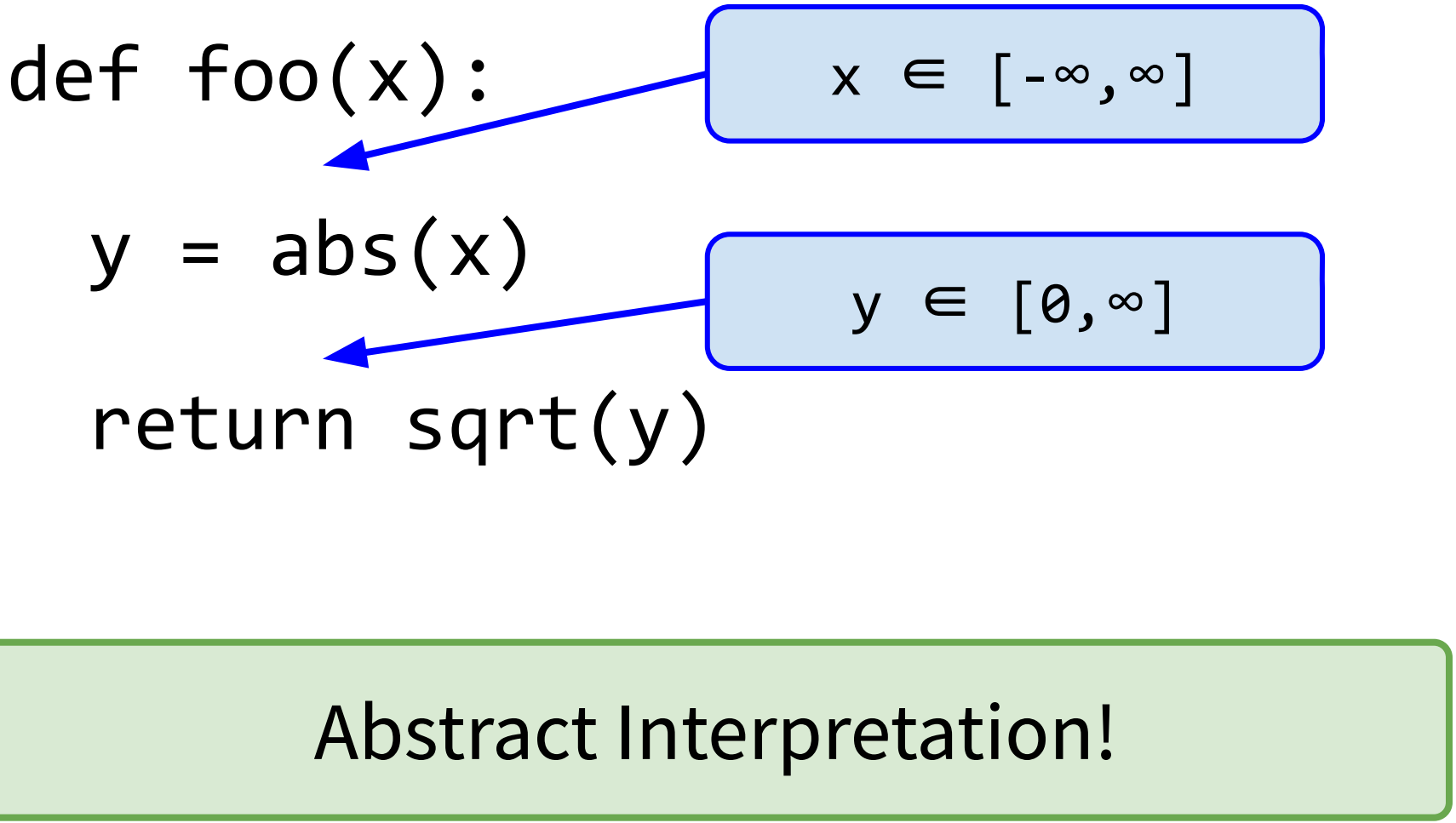
```
def foo(x):
    y = abs(x)
    return sqrt(y)
```

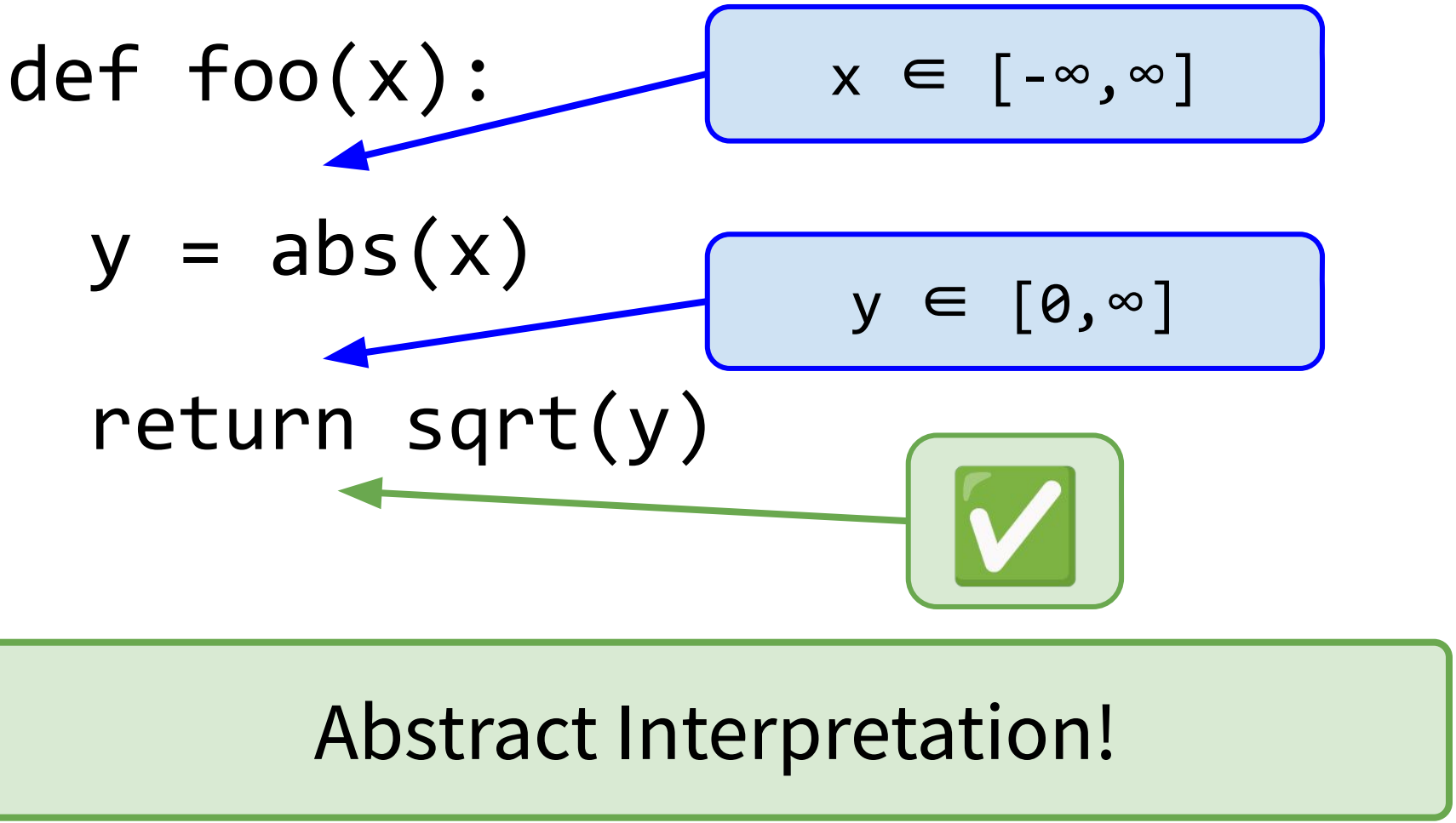
Abstract Interpretation!

```
def foo(x):
  y = abs(x)
  return sqrt(y)
```



Abstract Interpretation!


def foo(x):
x ∈ [-∞,∞]
y = abs(x)
y ∈ [0,∞]
return sqrt(y)
Abstract Interpretation!


def foo(x):
x ∈ [-∞,∞]
y = abs(x)
y ∈ [0,∞]
return sqrt(y)
✅
Abstract Interpretation!

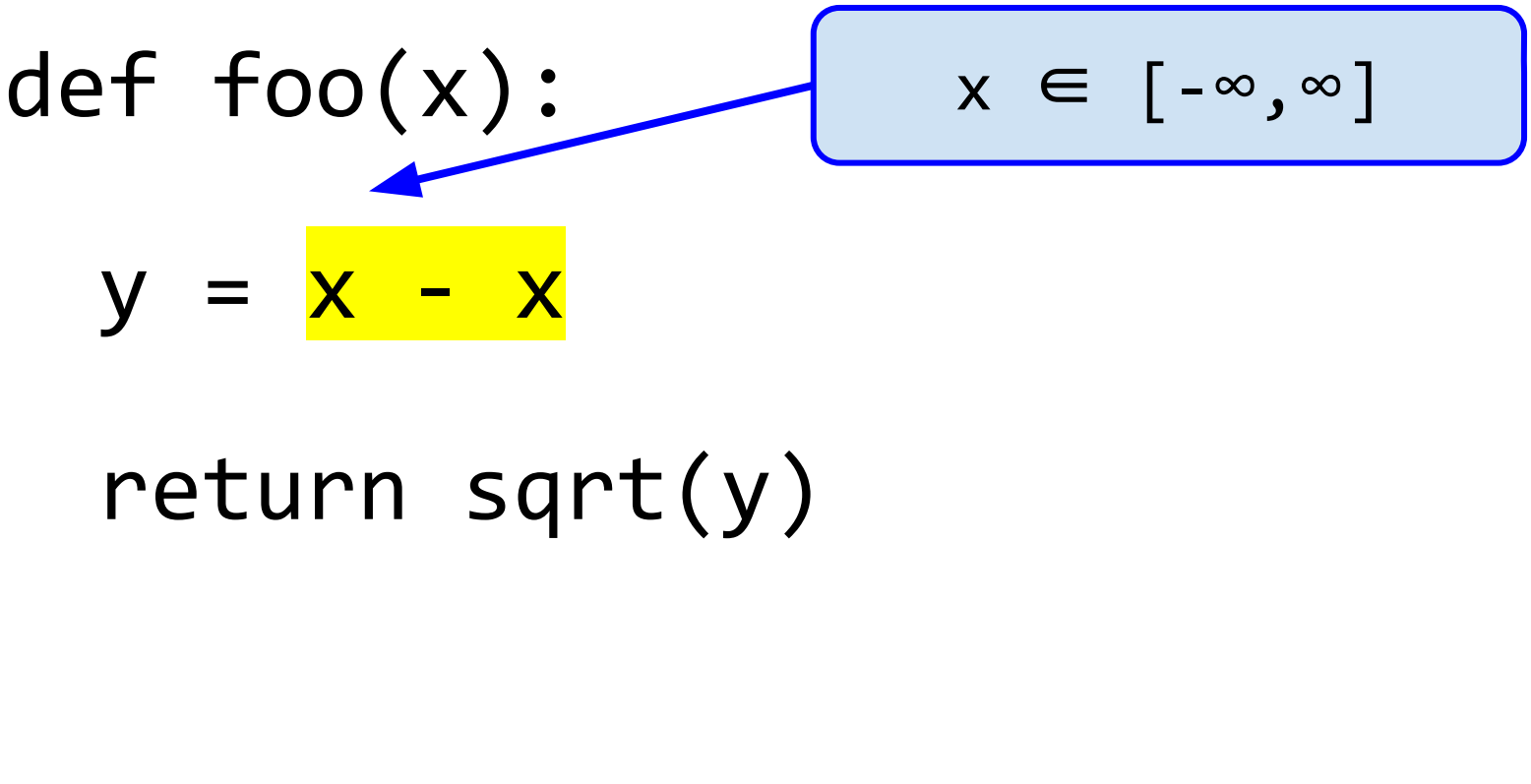
```
def foo(x):
  y = abs(x)
  return sqrt(y)
```

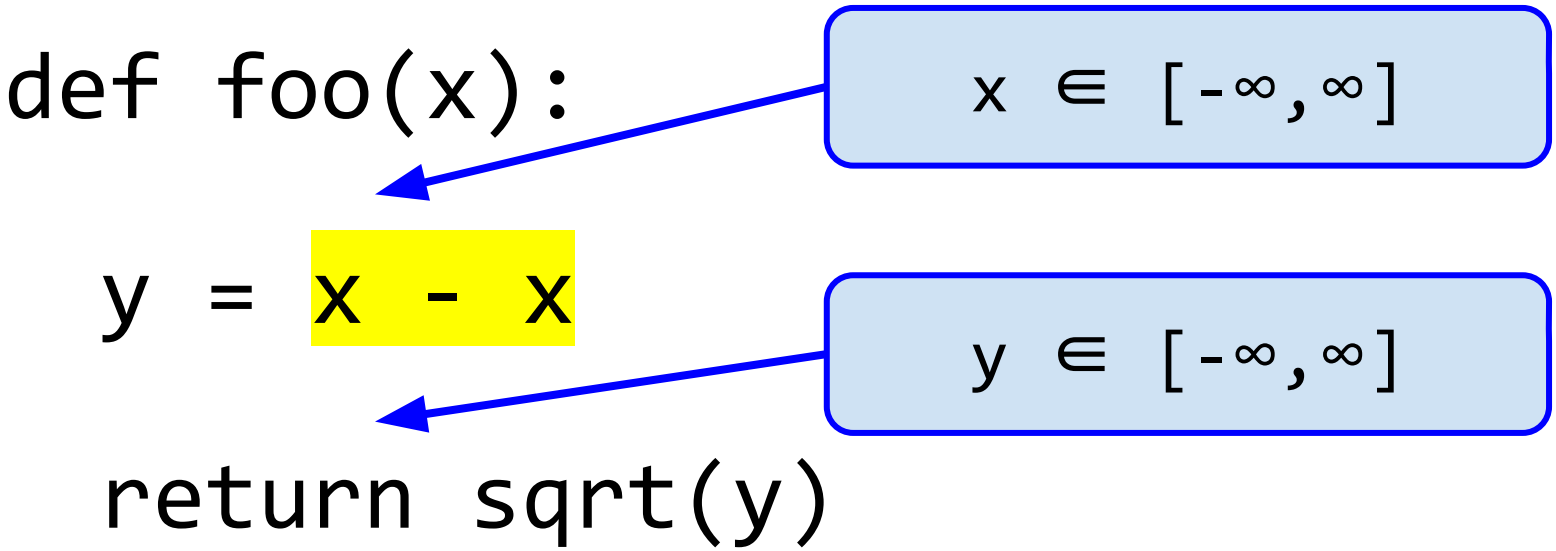
$x \in [-\infty, \infty]$

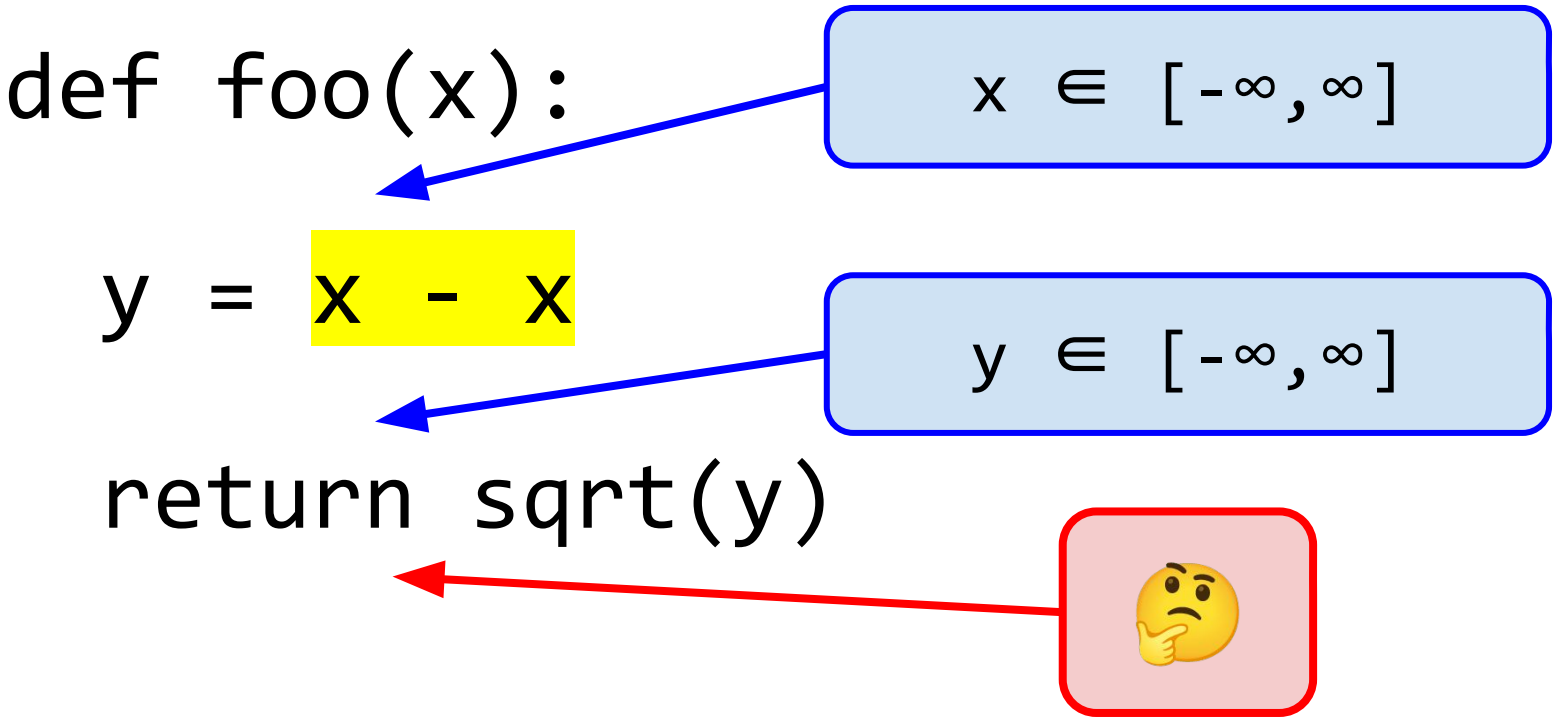
$y \in [0, \infty]$

BUT: abstraction can hide crucial facts!

```
def foo(x):
    y = x - x
    return sqrt(y)
```


def foo(x):
y = x - x
return sqrt(y)
x ∈ [-∞,∞]

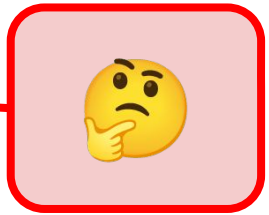

def foo(x):
y = x - x
return sqrt(y)
x ∈ [-∞,∞]
y ∈ [-∞,∞]


def foo(x):
x ∈ [-∞,∞]
y = x - x
y ∈ [-∞,∞]
return sqrt(y)
🤔

```
def foo(x):
    y = x - x
    return sqrt(y)
```

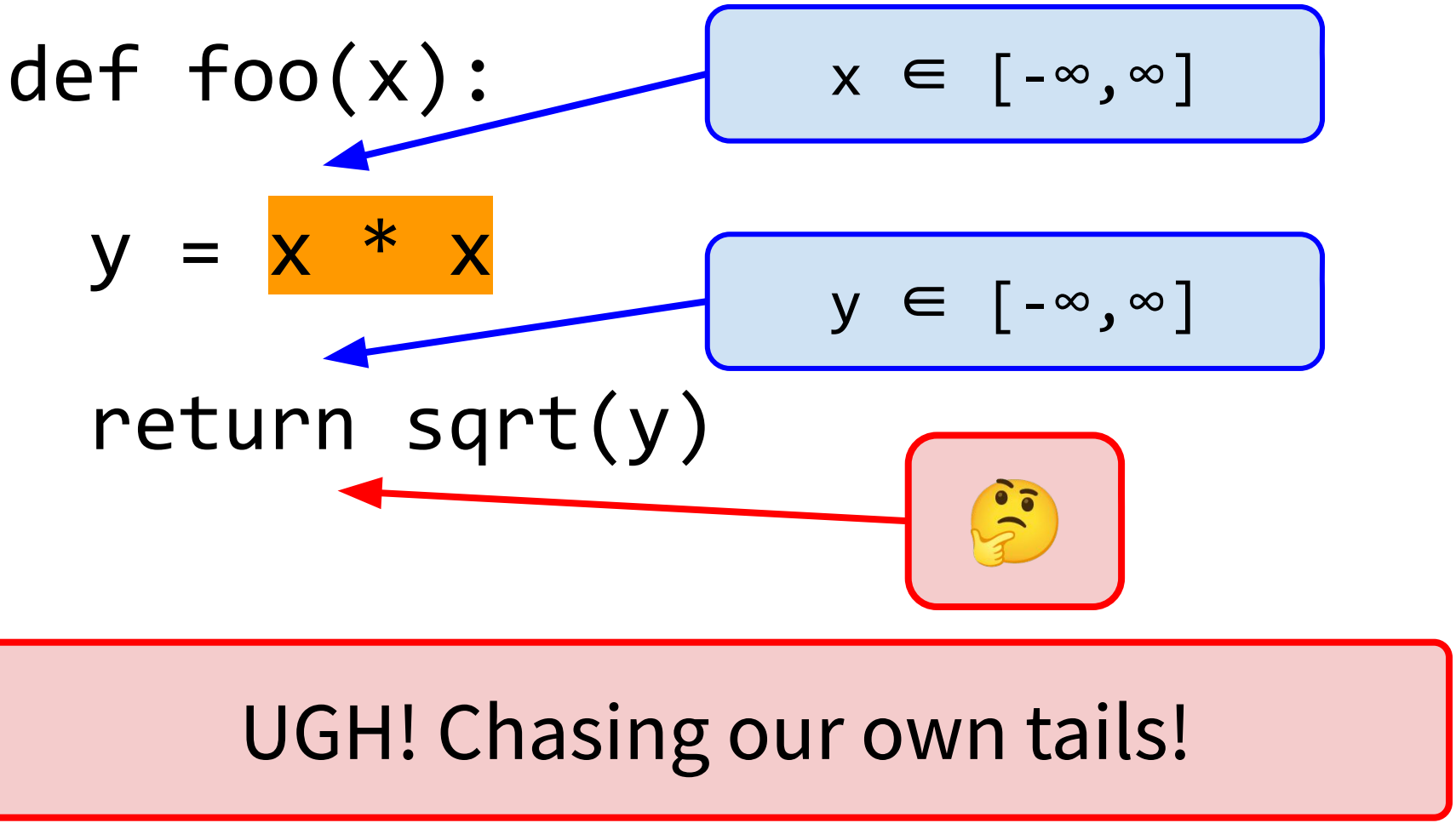
$x \in [-\infty,\infty]$

$y \in [-\infty,\infty]$



OK, just preprocess? $x - x \Rightarrow 0$

```
def foo(x):
    y = x * x
    return sqrt(y)
```


def foo(x):
x ∈ [-∞,∞]
y = x * x
y ∈ [-∞,∞]
return sqrt(y)
🤔
UGH! Chasing our own tails!

PL Problems

1. Phase Ordering 😭
2. Syntactic Brittleness 😭
3. …

PL Problems

1. Phase Ordering 😭
2. Syntactic Brittleness 😭
3. …

Keep simple + modular!

But make robust?

PL Problems

1. Phase Ordering 😭
2. Syntactic Brittleness 😭
3. …

Keep simple + modular!

But make robust?

DB techniques will come to our rescue!

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

Which rewrite? When?

Useful

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

      x / x = 1

      x * 1 = x
```

Less Useful

```
x * 2 = x << 1

x * y = y * x

    x = x * 1
```

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

Which rewrite? When?

Equality Saturation

Try applying all the rules in every order!

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

Which rewrite? When?

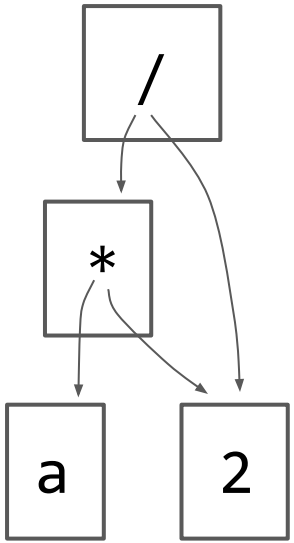
Equality Saturation

Try applying **all** the rules in **every** order?!

E-graphs

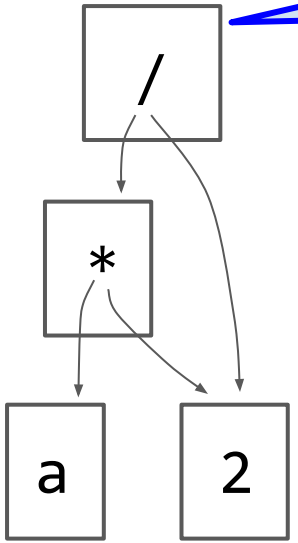
- Data structure from Greg Nelson's PhD thesis (1980)
- Used for congruence closure (Downey, Sethi, Tarjan 1980)
  - Intuition: union-find (Tarjan 1975) but function-aware
- Key for equality and uninterpreted funcs (EUF) theory in SMT
  - Intuition: the "glue" that connects other theories to SAT
- Historically: "baked in" to SMT solvers, no general libraries 😐

E-graphs

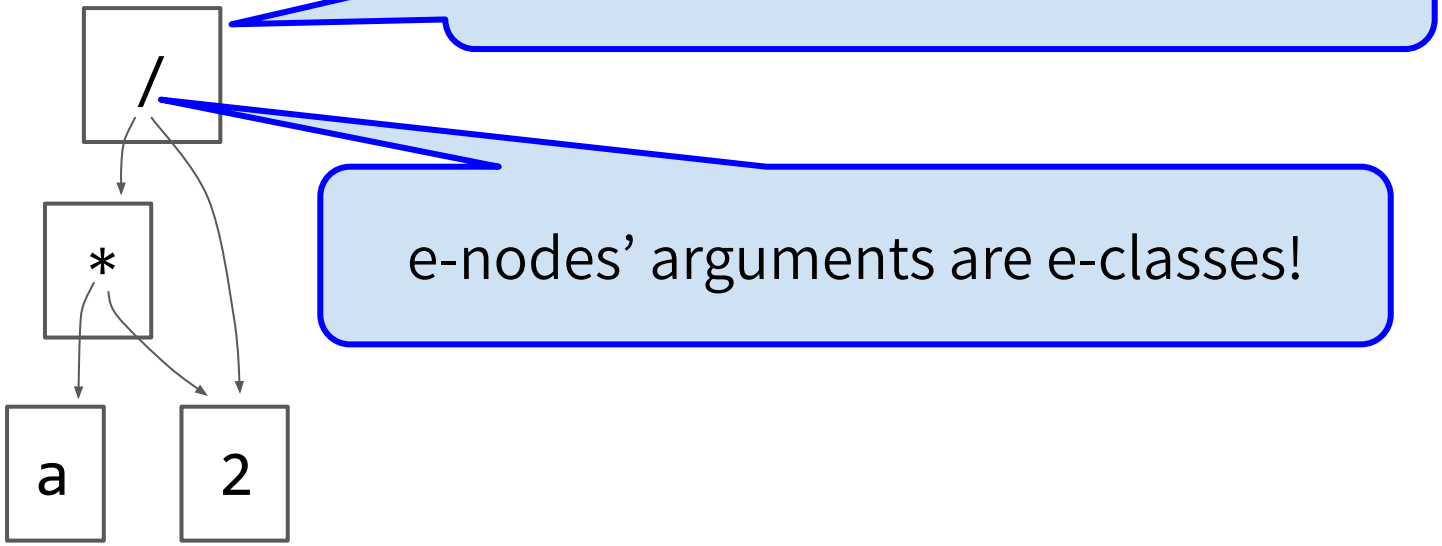


E-graphs

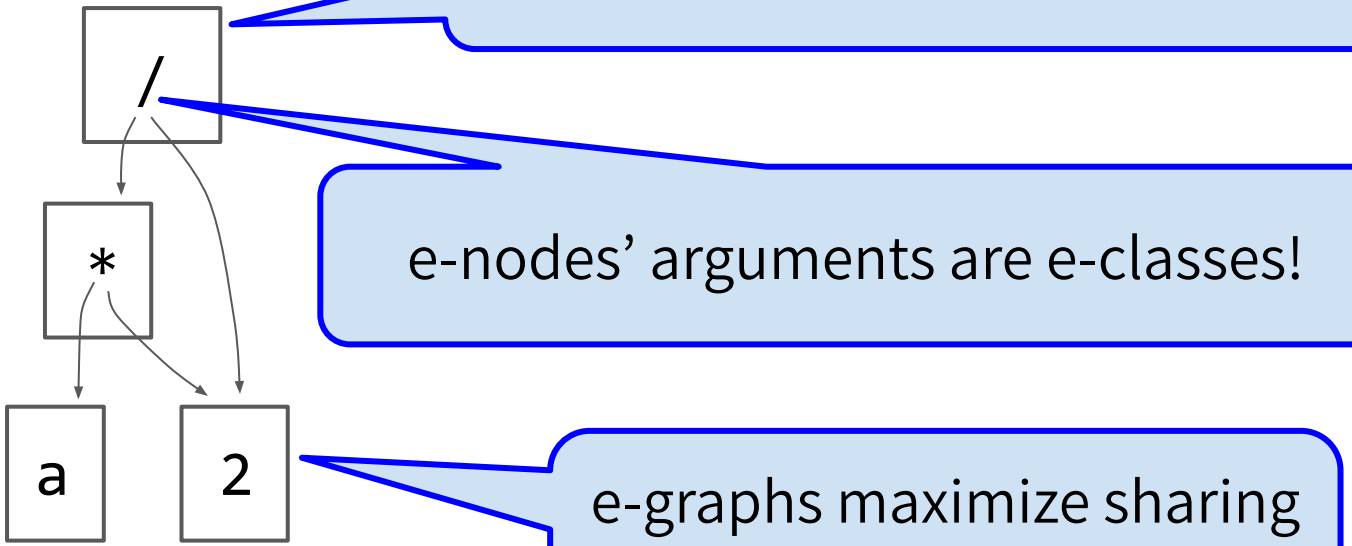
e-classes contain e-nodes (ops)



E-graphs


e-classes contain e-nodes (ops)
/
e-nodes’ arguments are e-classes!
*
a
2

E-graphs

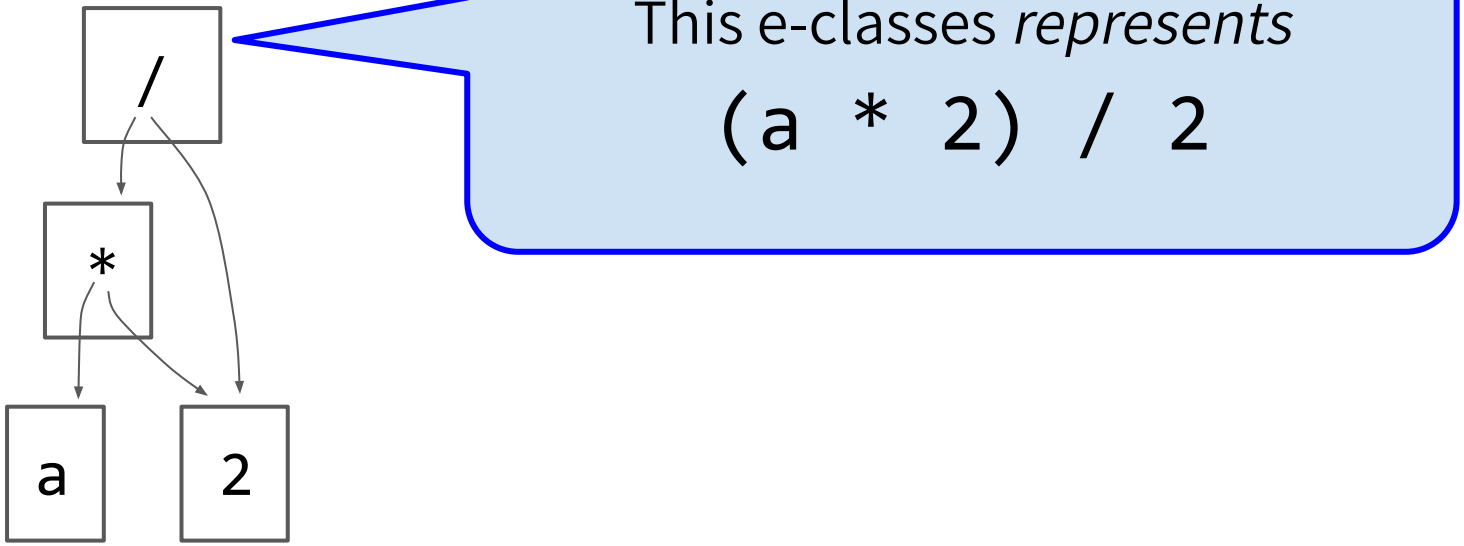


e-classes contain e-nodes (ops)

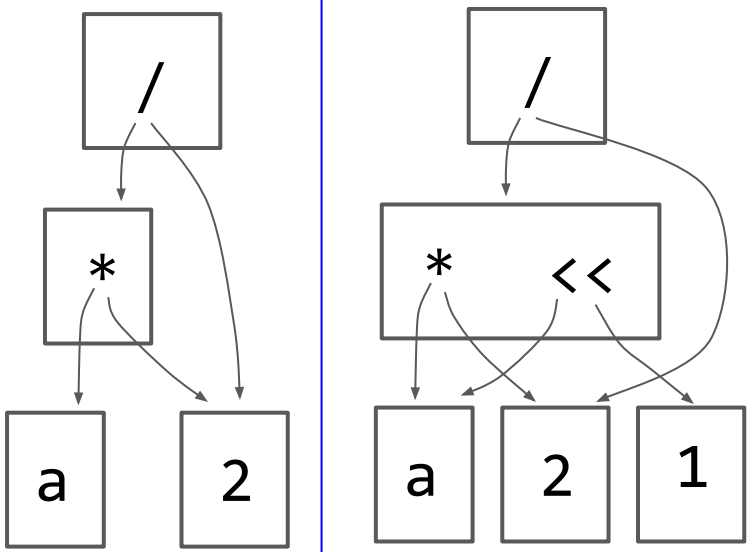
e-nodes' arguments are e-classes!

e-graphs maximize sharing
(no copies of same e-node)

E-graphs

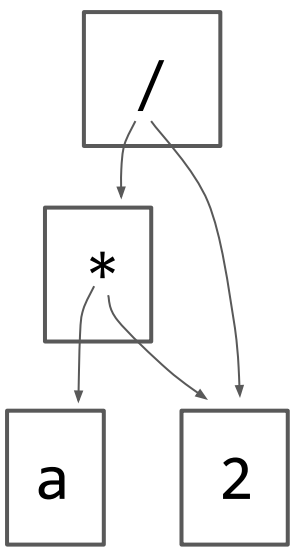


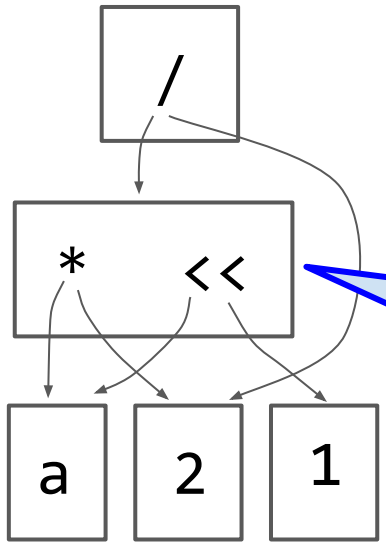
E-graphs: applying rewrite rules



x * 2 → x << 1

E-graphs: applying rewrite rules



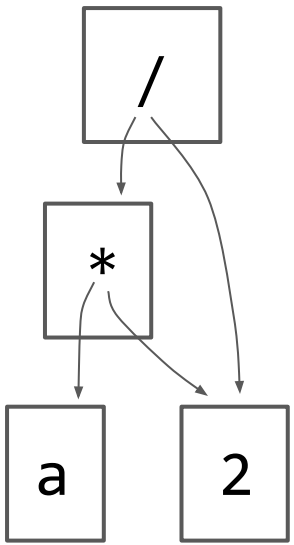


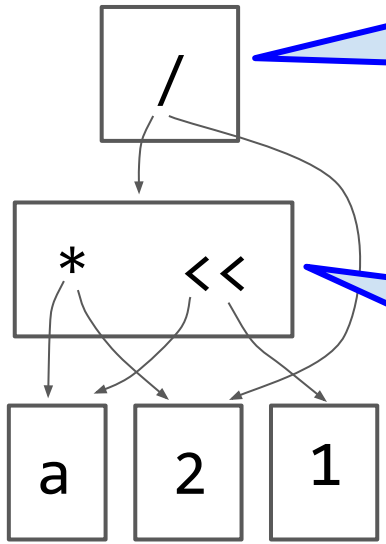
This e-class *represents*

a*2 **and** a<<1

x * 2 → x << 1

E-graphs: applying rewrite rules



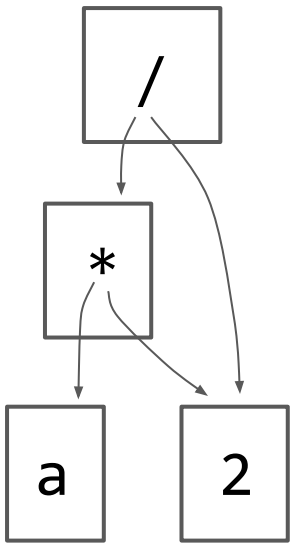


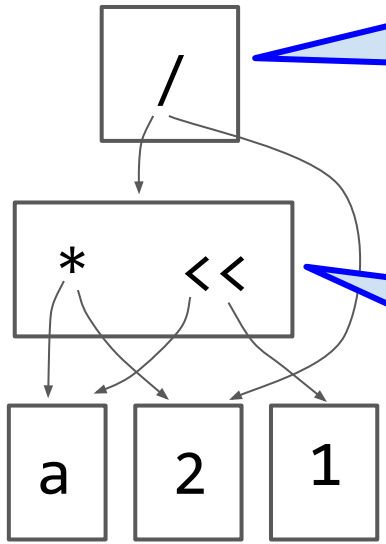
This e-class *represents*
`(a*2)/2` **and** `(a<<1)/2`

This e-class *represents*
`a*2` **and** `a<<1`

x * 2 → x << 1

E-graphs: applying rewrite rules



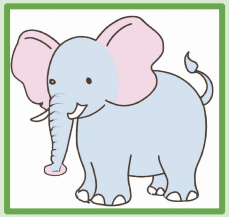


This e-class *represents*
`(a*2)/2` **and** `(a<<1)/2`

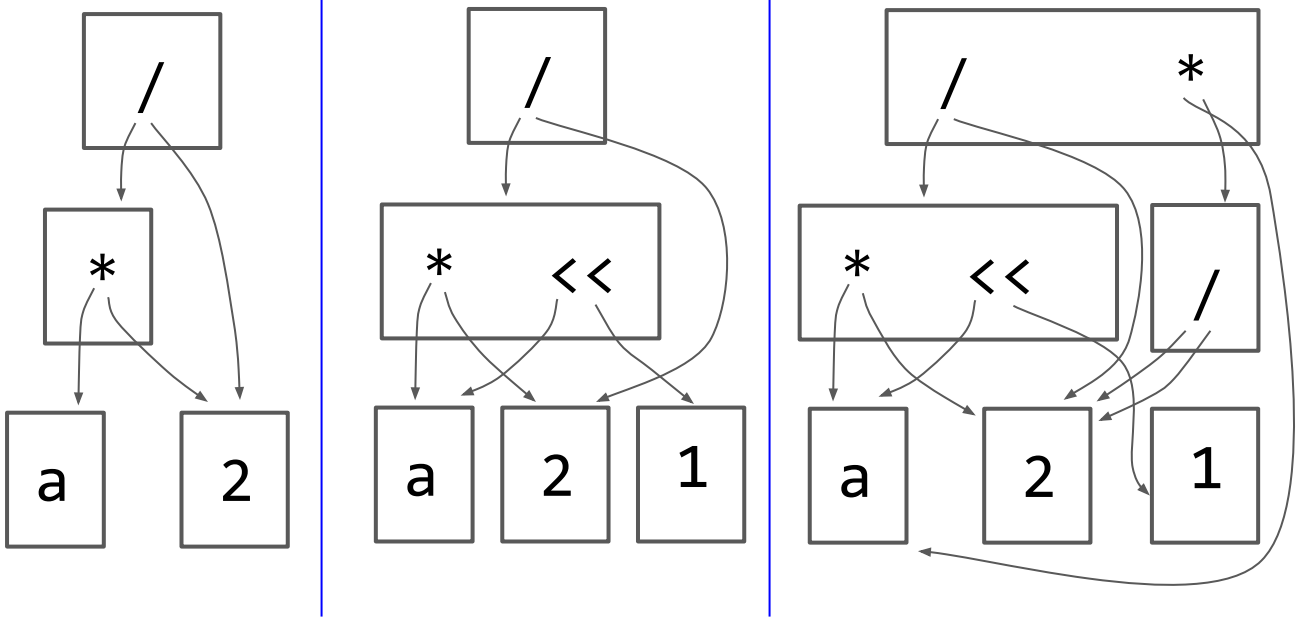
This e-class *represents*
`a*2` **and** `a<<1`

x * 2 → x << 1

E-graphs never forget.
Rewrites don't lose info!

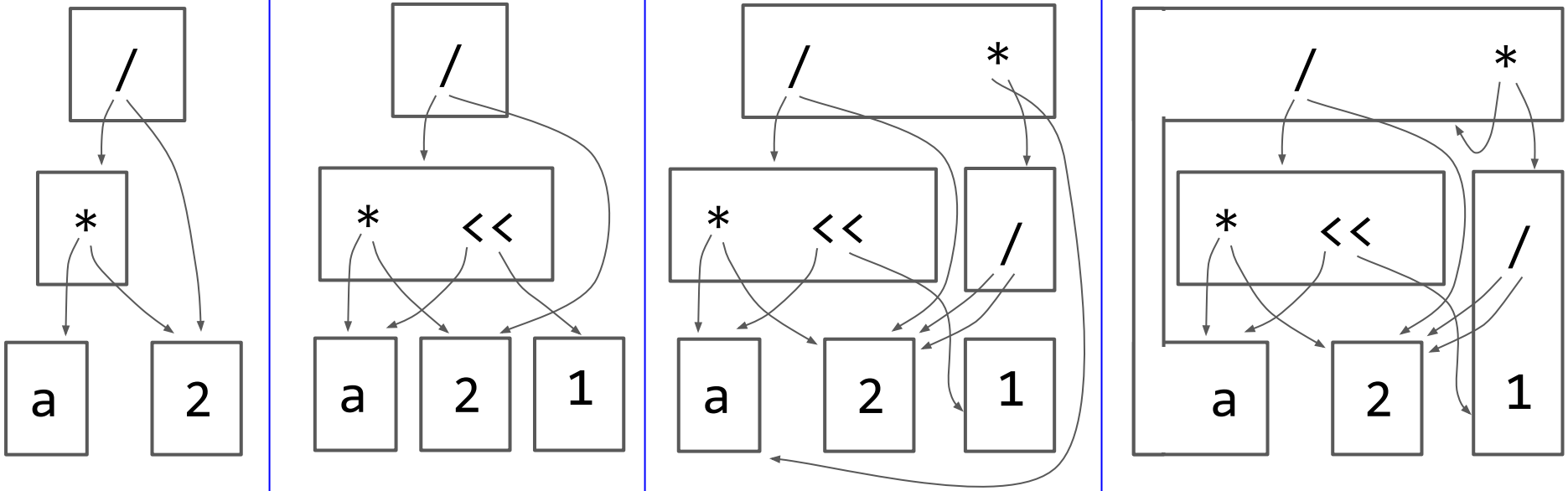


E-graphs: applying rewrite rules



`x * 2 → x << 1` `(x*y)/z → x*(y/z)`

E-graphs: applying rewrite rules



x * 2 → x << 1

(x*y)/z → x*(y/z)

x / x → 1

x * 1 → x

E-graphs: compact representation

Rewrites can **shrink** e-graphs!

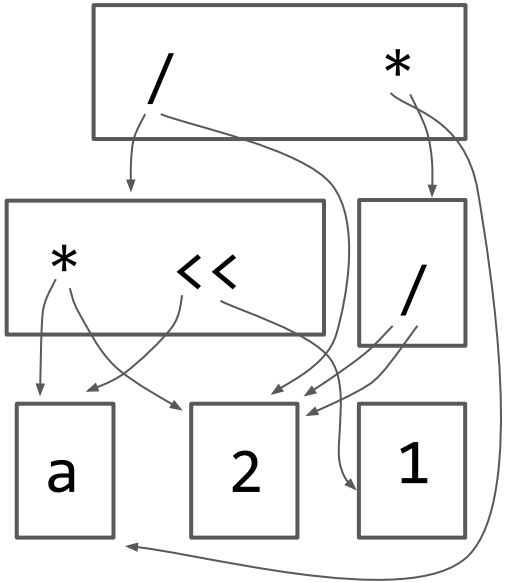
- 6 → 5 eclasses

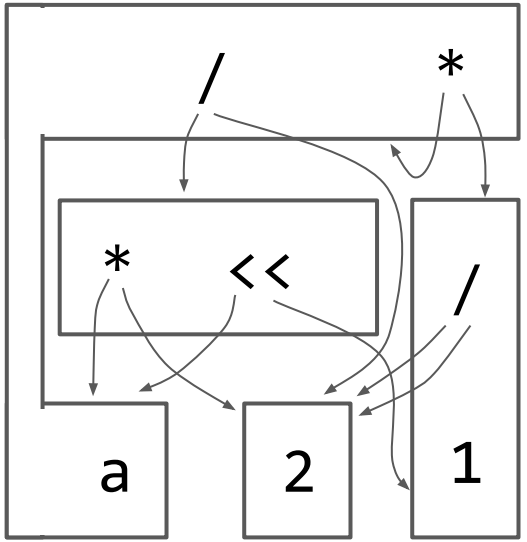
E-graphs can represent ∞ terms

- `a, a * 1, a * 1 * 1, …`

E-graphs can "*saturate*"

- learn all derivable eqs ✅





x / x → 1

x * 1 → x

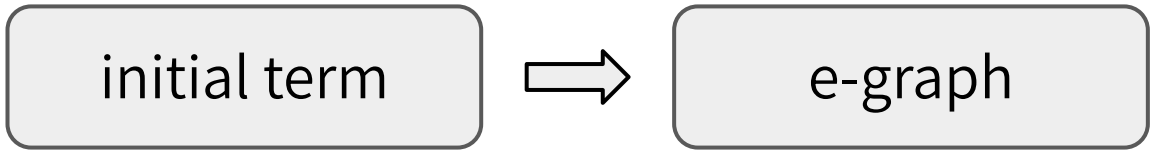
Equality Saturation

- Technique first used in Denali (Joshi, Nelson, Randall 2002)
  - Optimizes straight-line assembly kernels for the DEC Alpha
- Extended to loops in Peggy [POPL 2009]
  - Coined term “Equality Saturation”
  - Coinductive stream operators for algebraic loop rewrites
  - Used Rete algo from expert systems

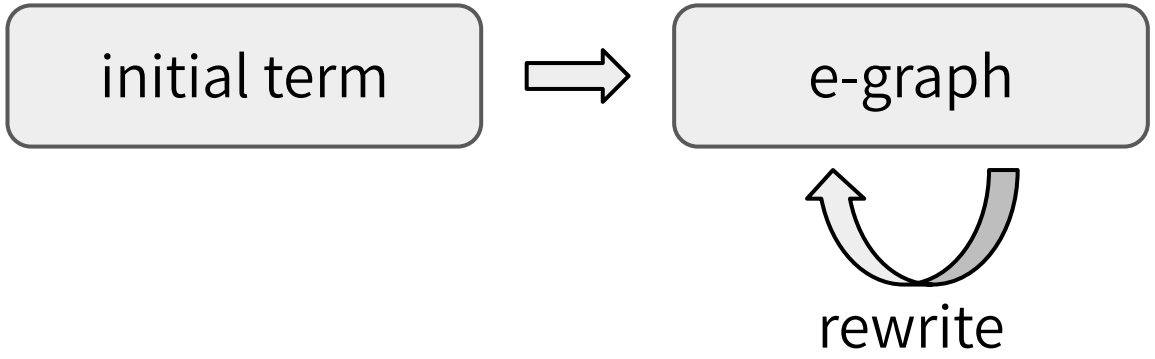
Equality Saturation

initial term

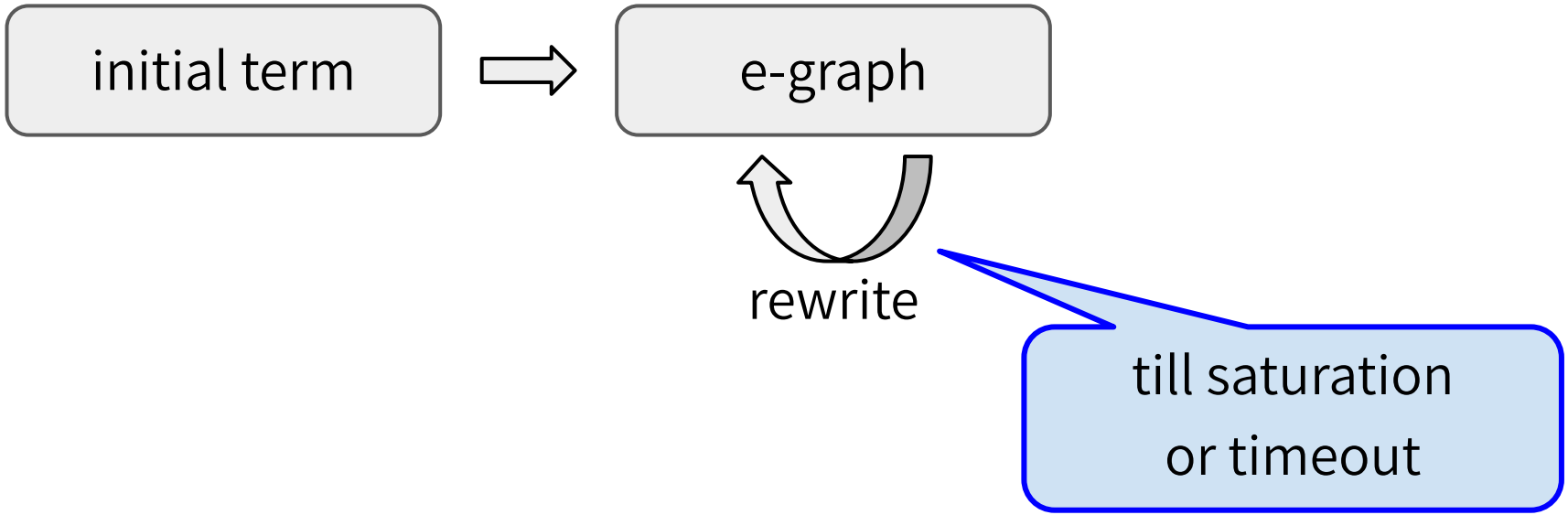
Equality Saturation



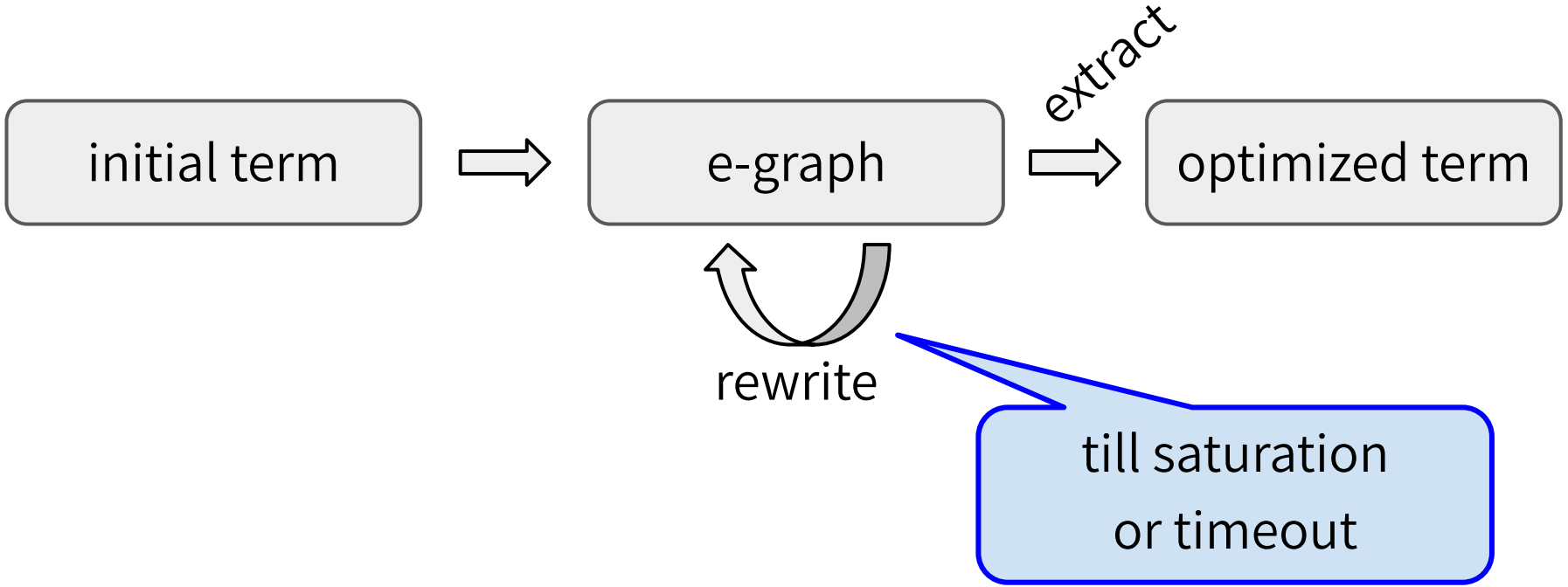
Equality Saturation



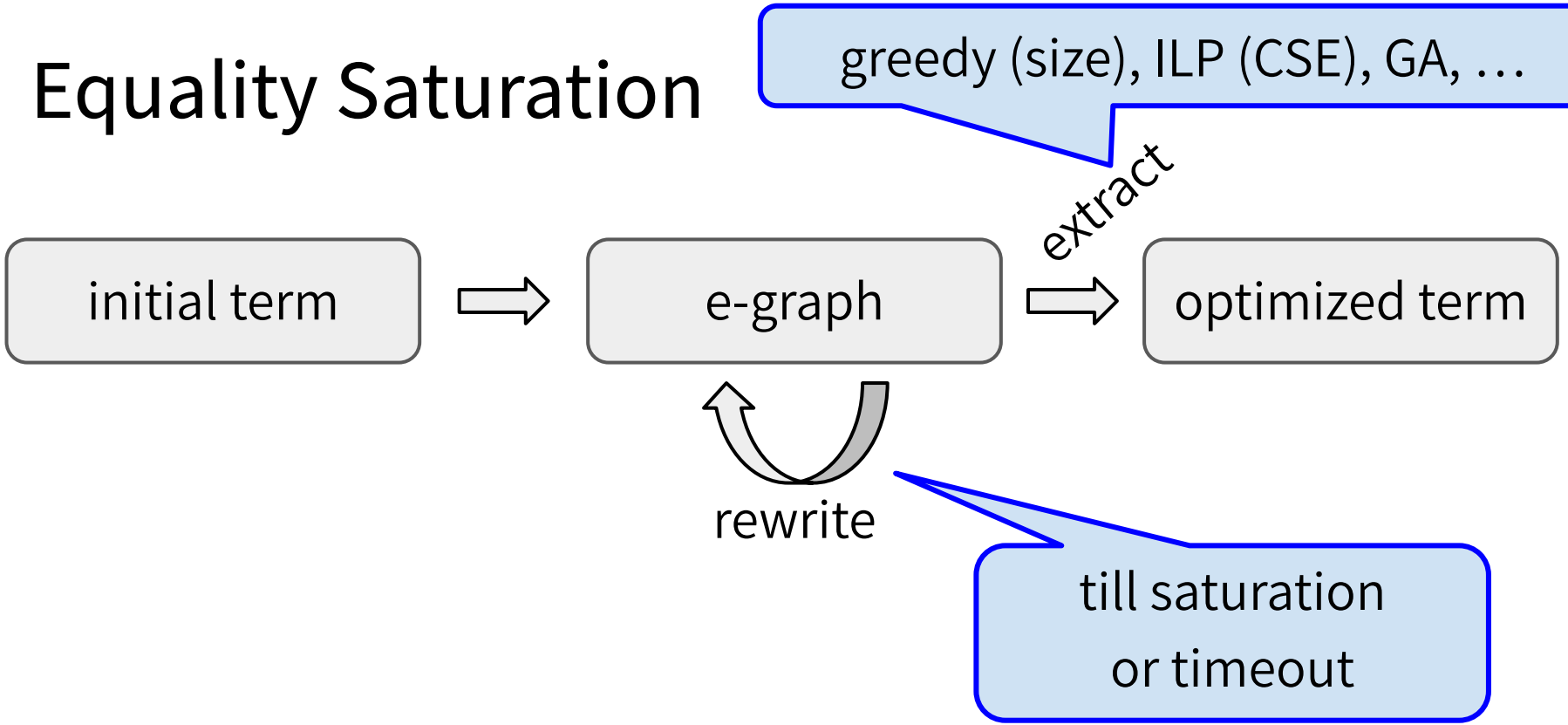
Equality Saturation



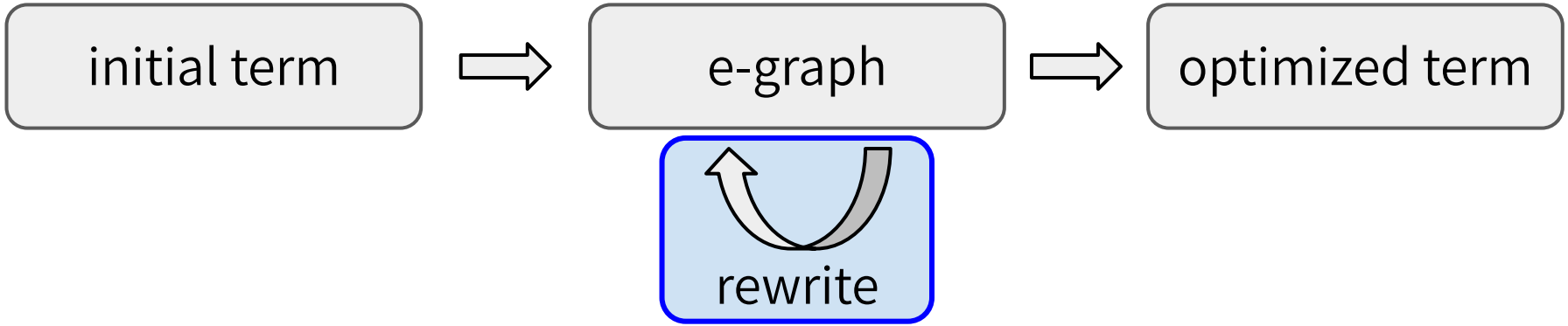
Equality Saturation



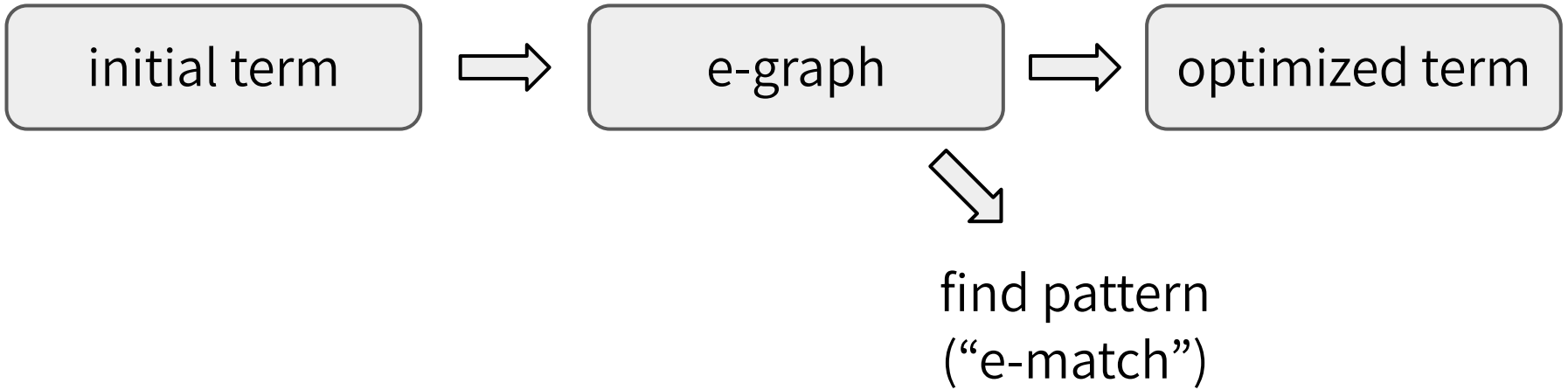
Equality Saturation



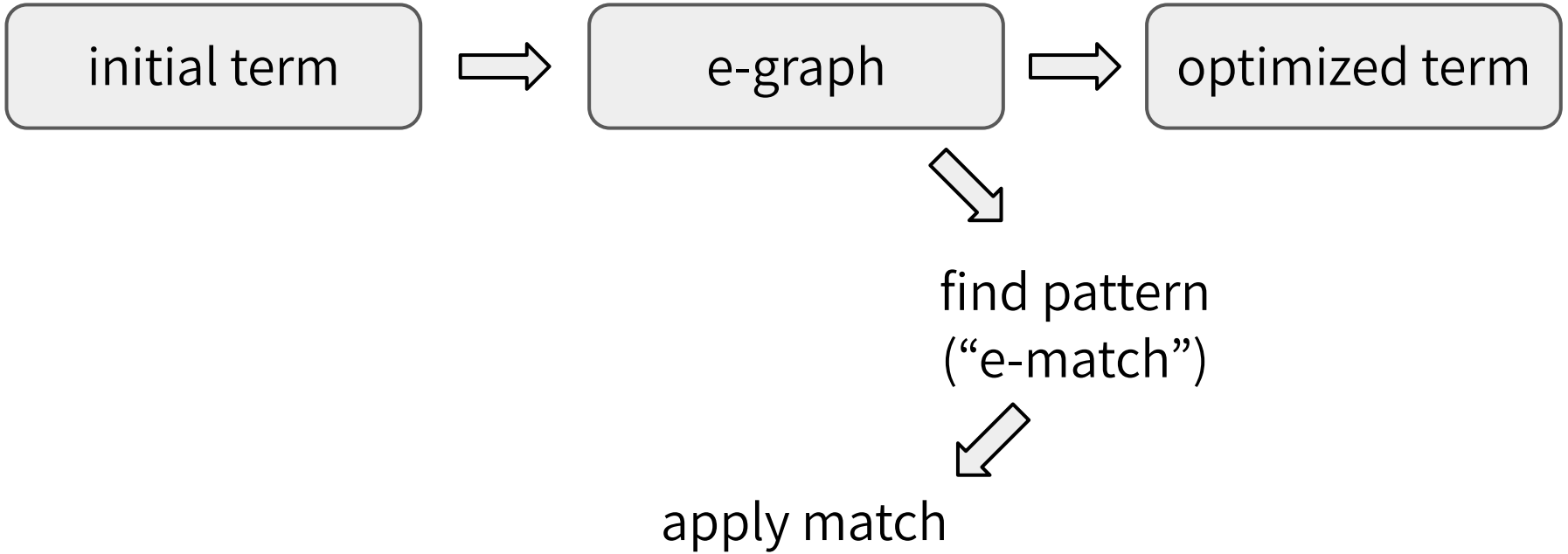
Equality Saturation



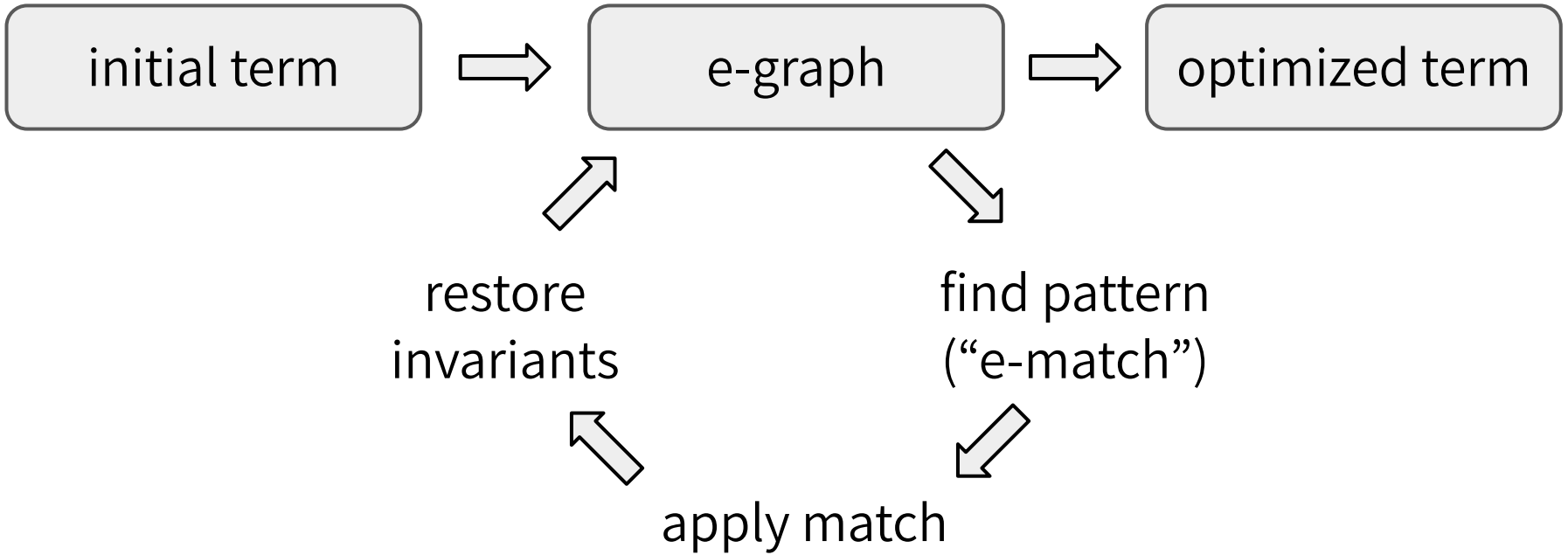
Equality Saturation



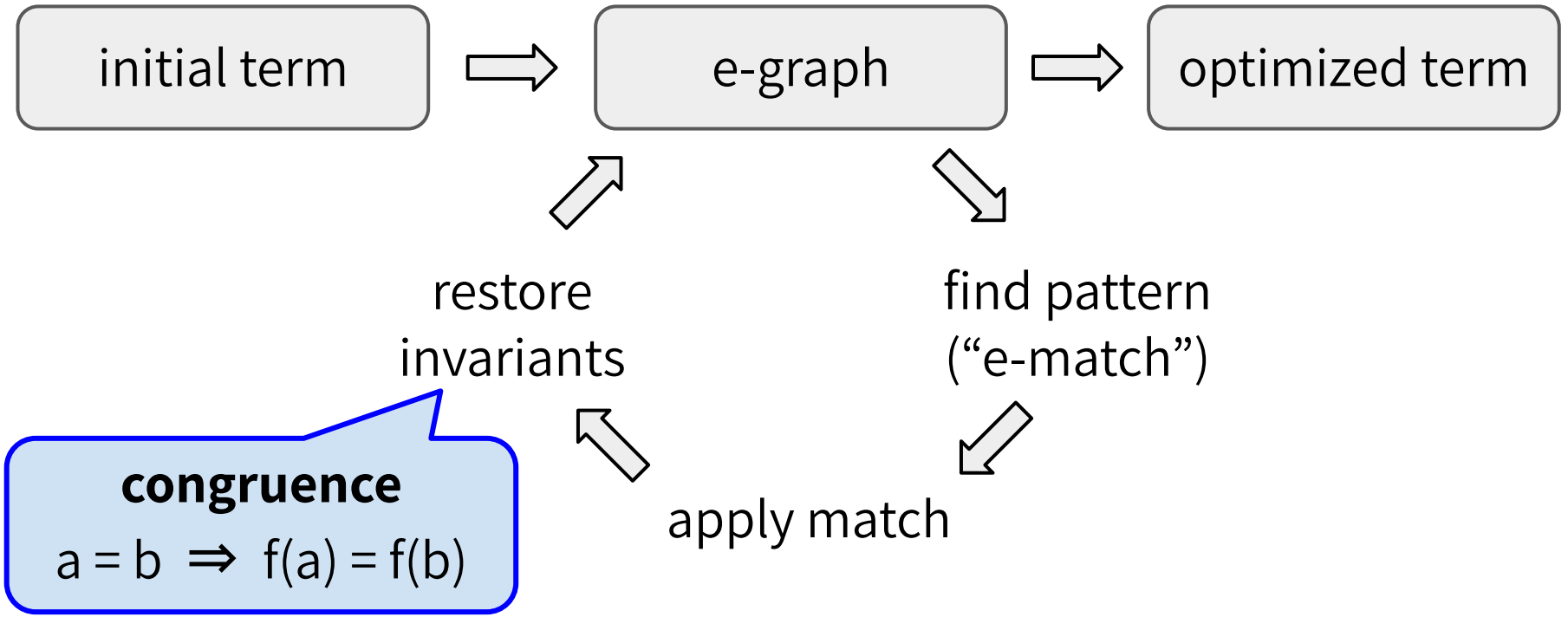
Equality Saturation



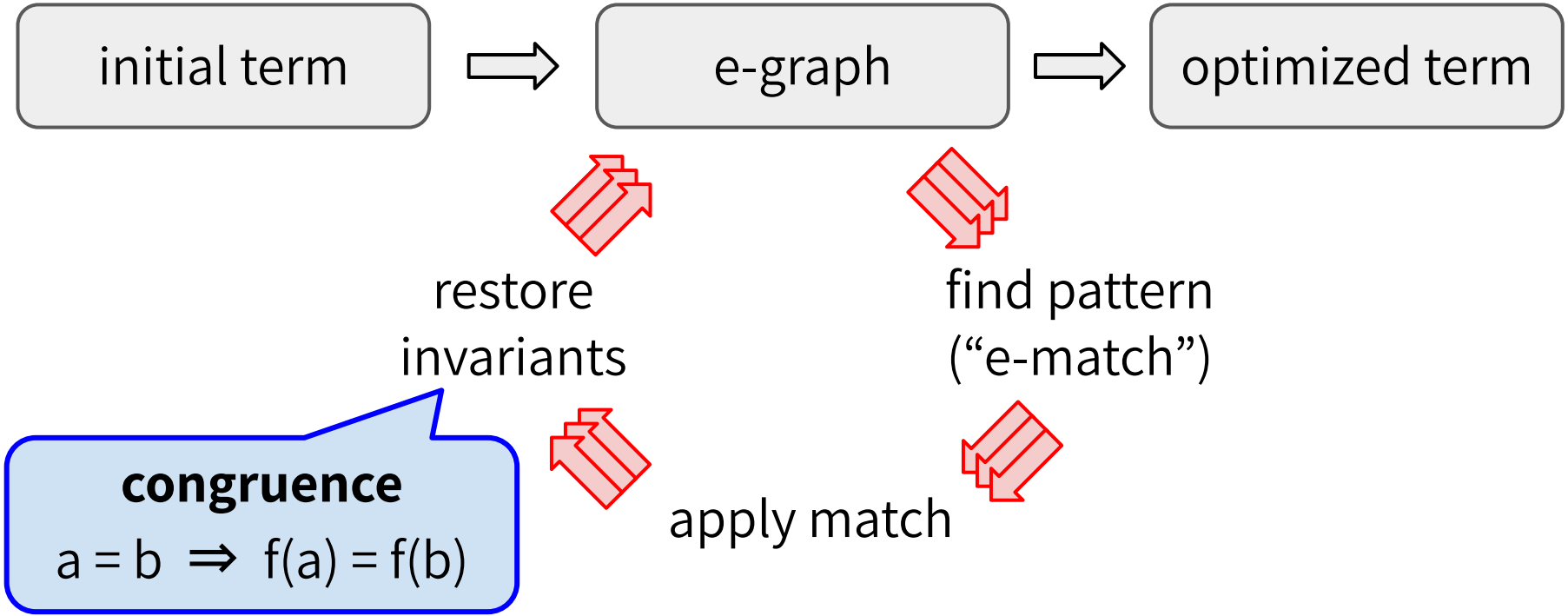
Equality Saturation



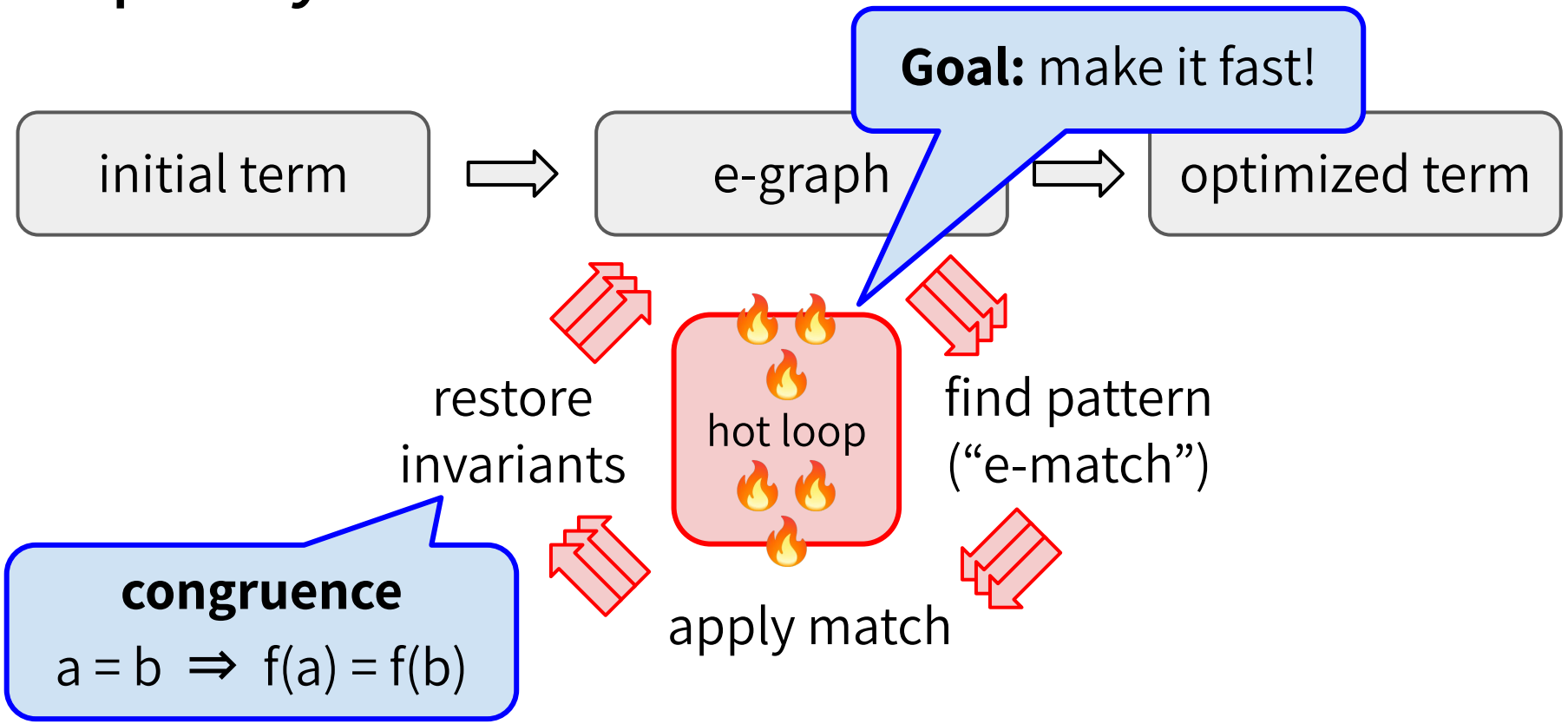
Equality Saturation



Equality Saturation



Equality Saturation



egg: Fast and Flexible EqSat [POPL 2021]

```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():
    matches = []
    for rw in rewrites:
      for (subst, ec) in egraph.ematch(rw.lhs):
        matches.append((rw, subst, ec))
    for (rw, subst, ec) in matches:
      ec2 = egraph.add(rw.rhs.subst(subst))
      egraph.merge(ec, ec2)
    egraph.rebuild()

  return egraph.extract_best()
```

batch reads

batch writes (invariants broken)

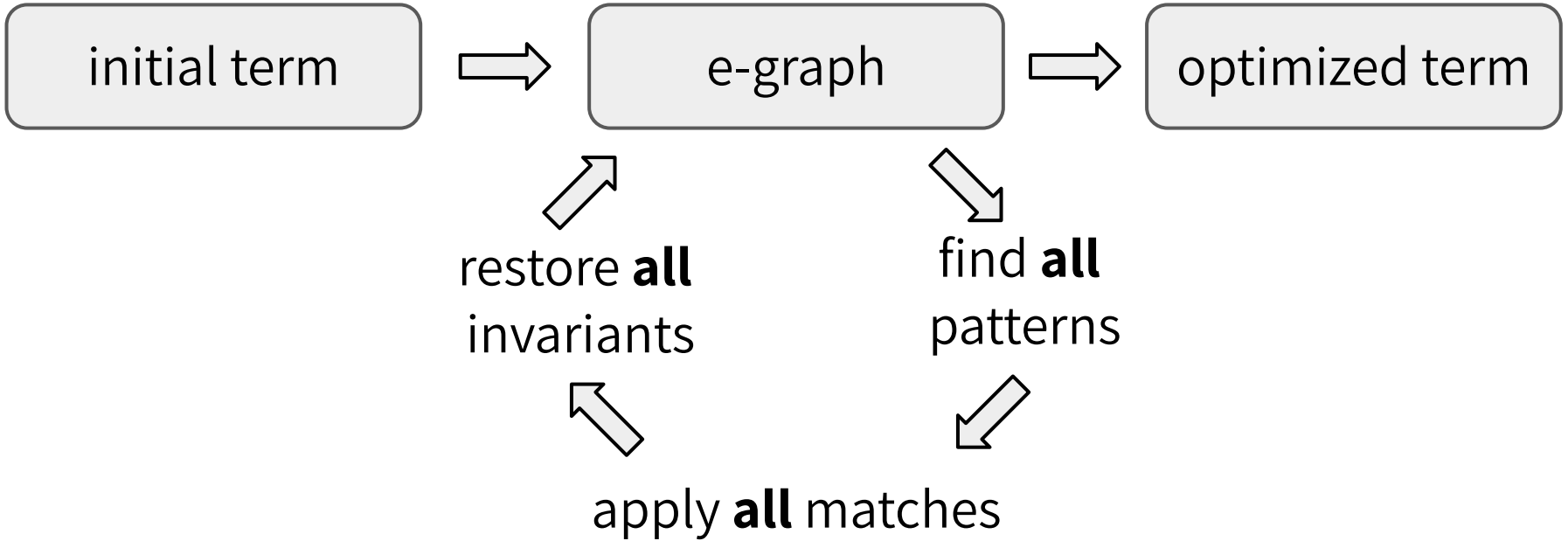
invariants restored once per iteration

congruence

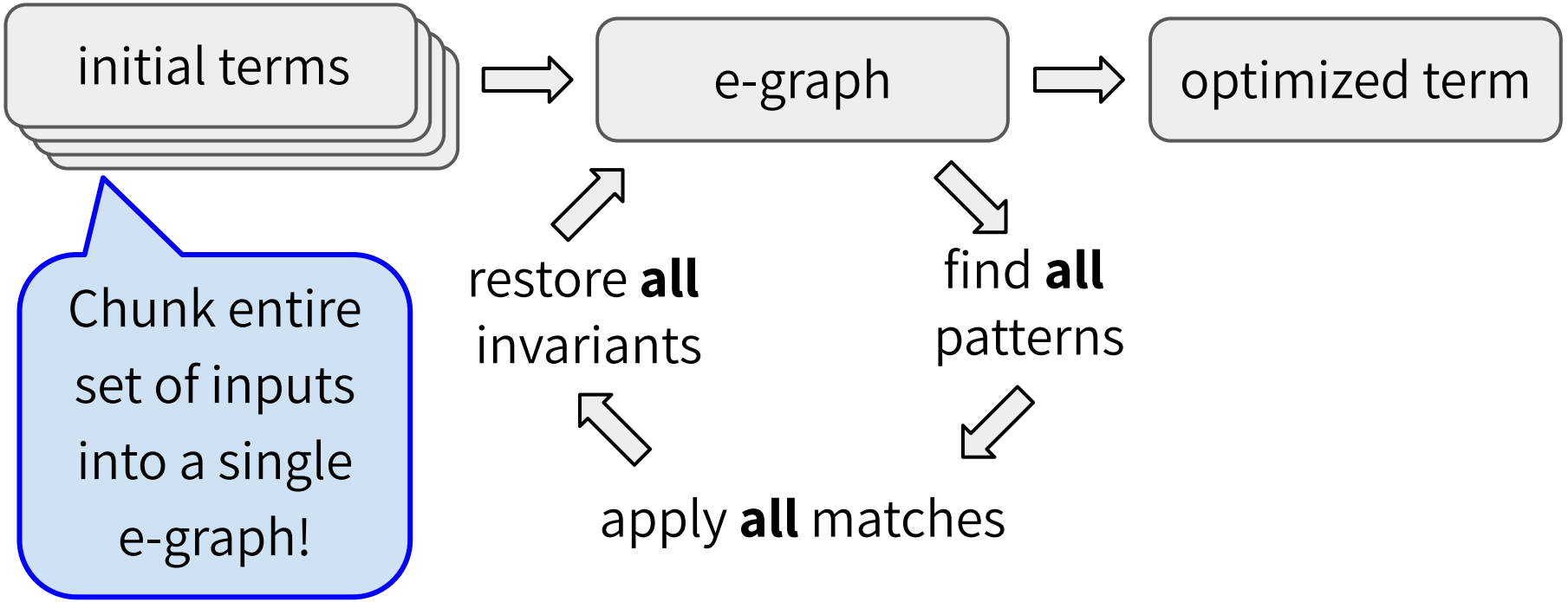
$a = b \Rightarrow f(a) = f(b)$

Adapted from Downey, Sethi, Tarjan 1980

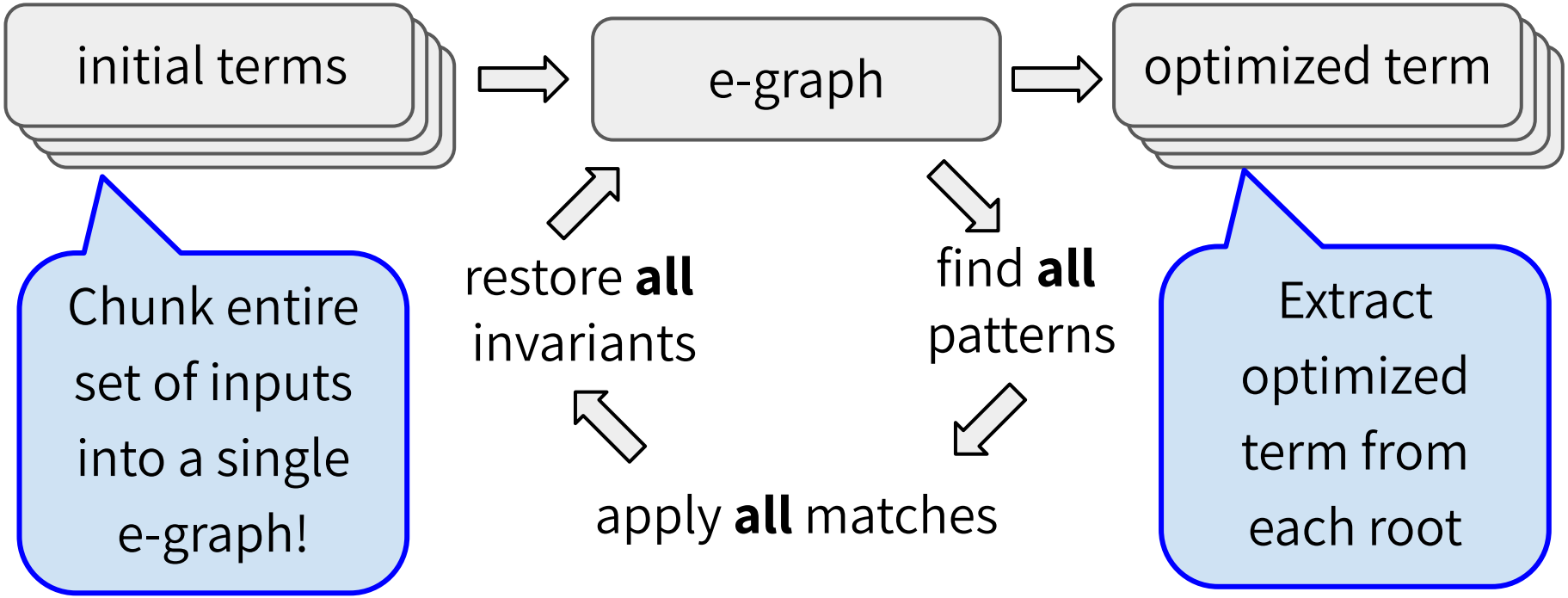
More amortization via *batching* in egg



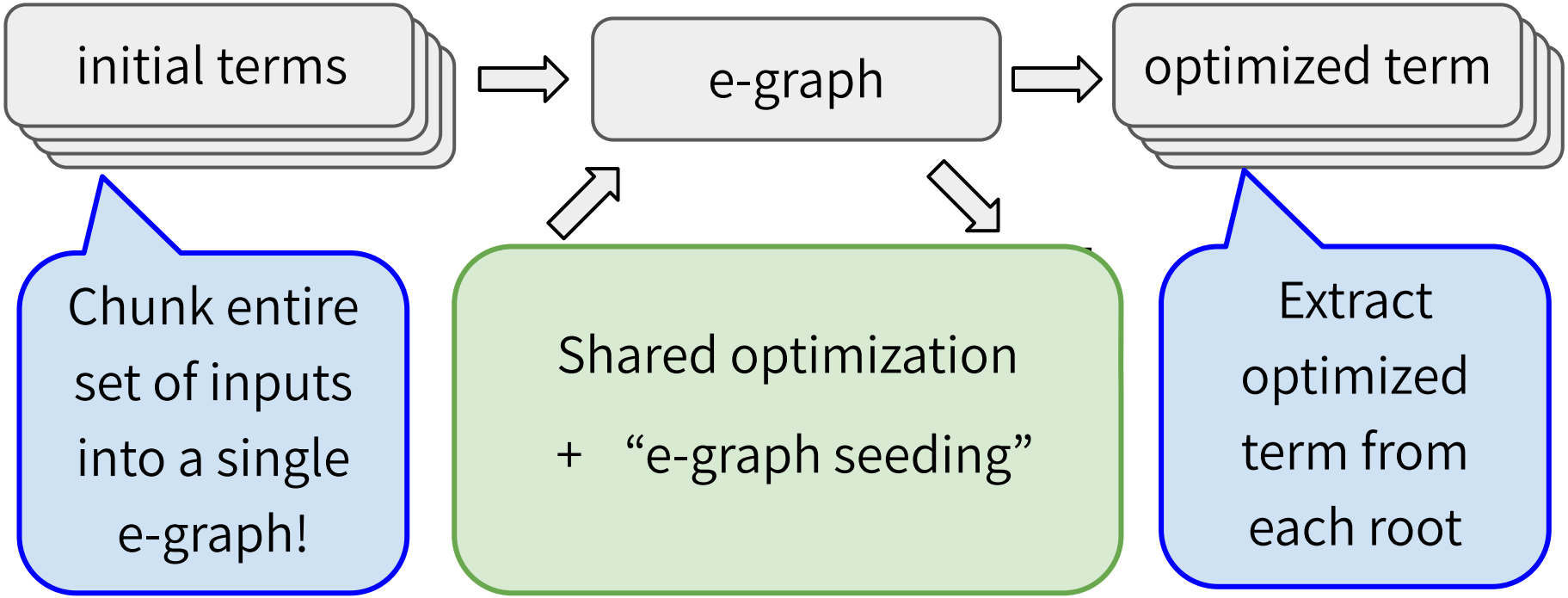
More amortization via *batching* in egg



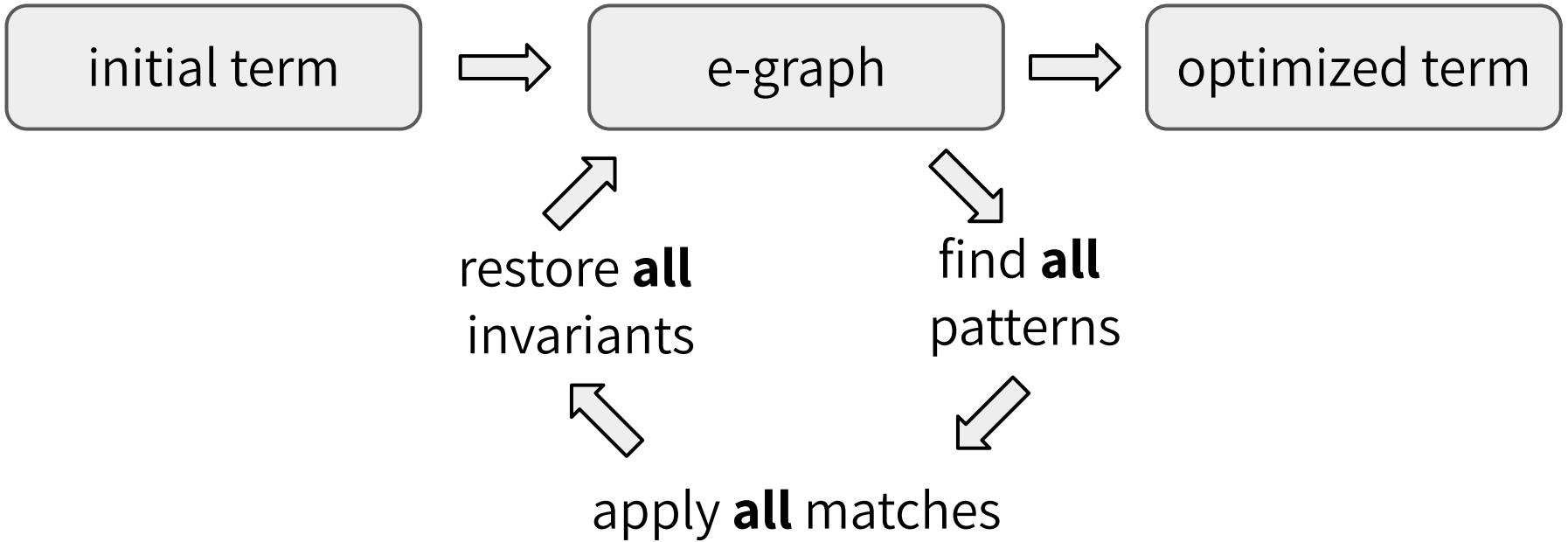
More amortization via *batching* in egg



More amortization via *batching* in egg

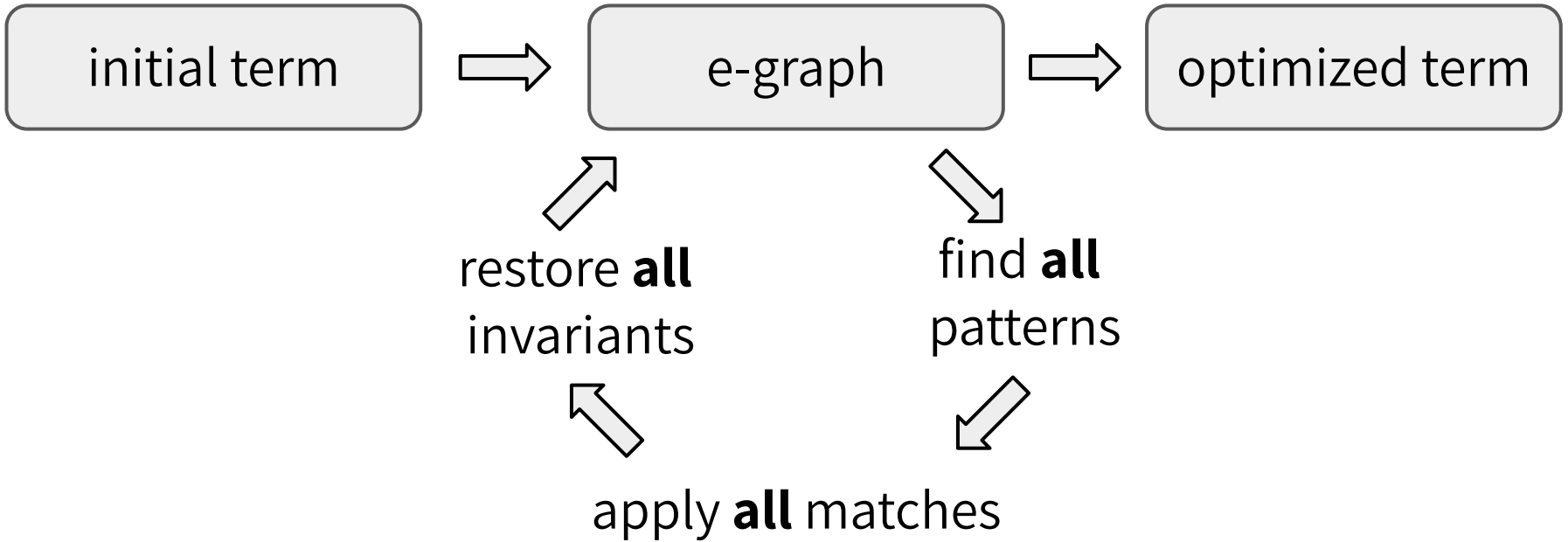


egg's Equality Saturation

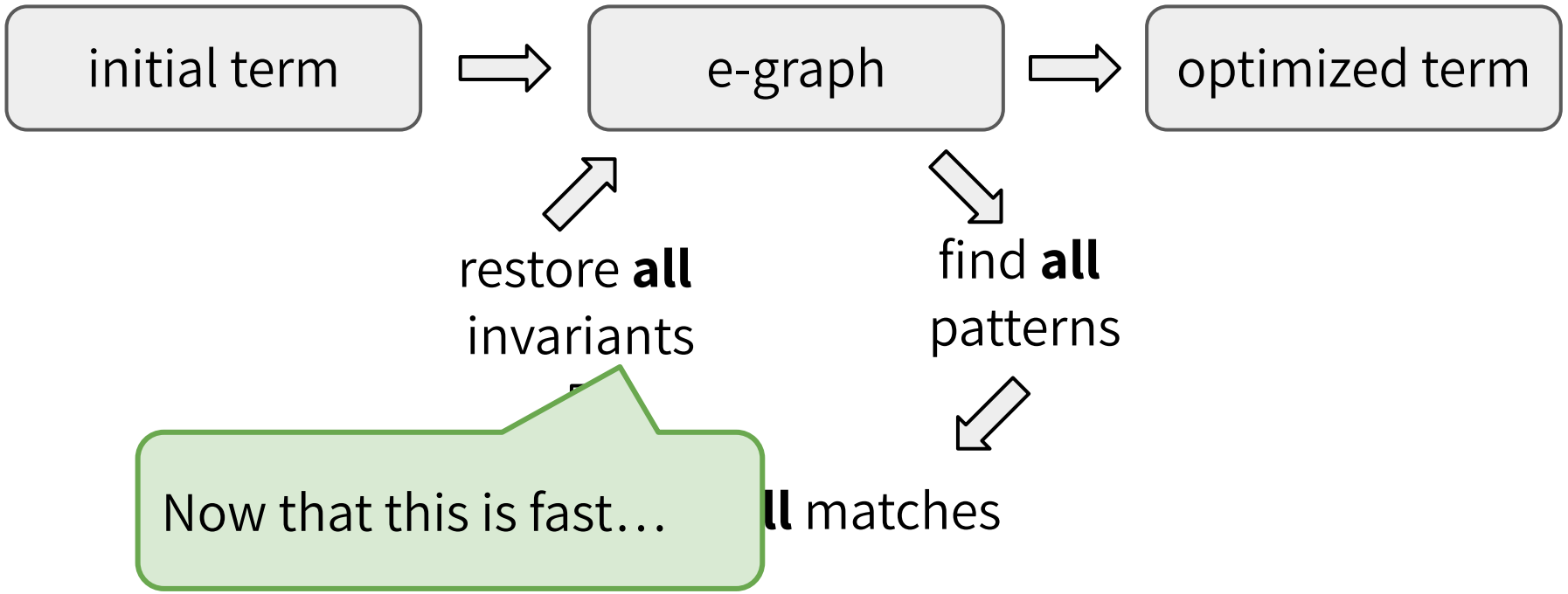


egglog : EqSat + Datalog

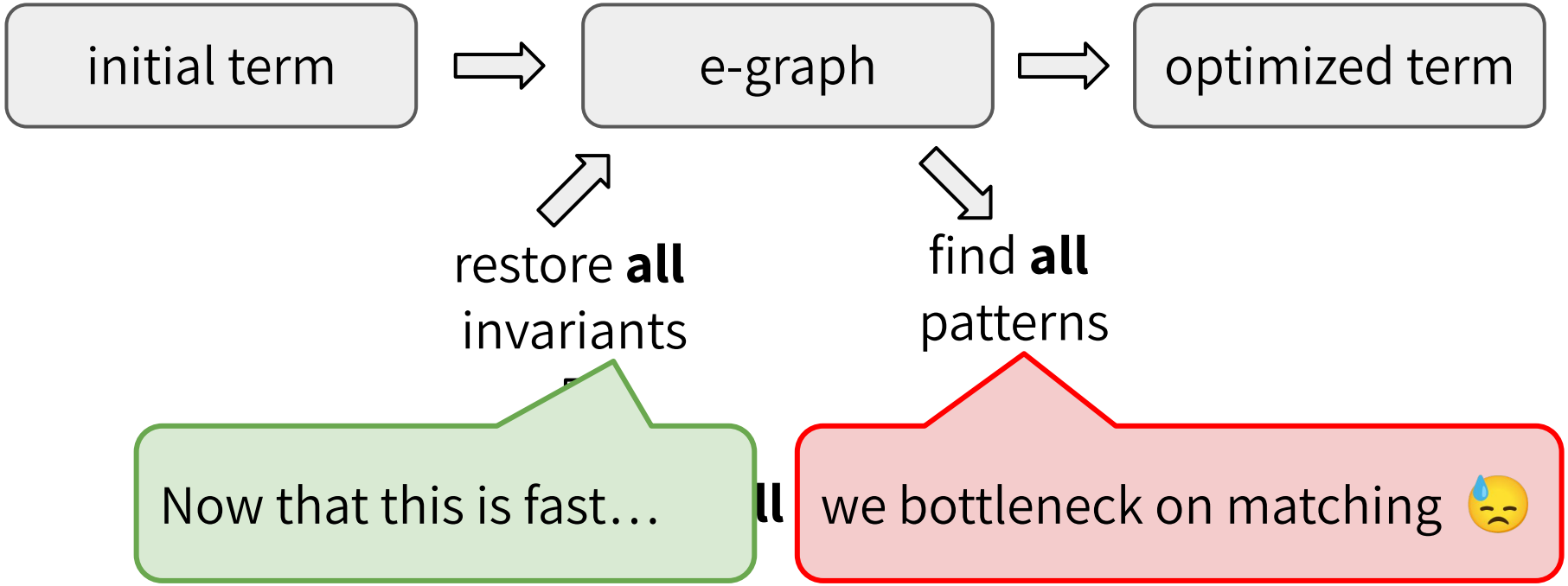
egg's Equality Saturation



egg's Equality Saturation



egg's Equality Saturation

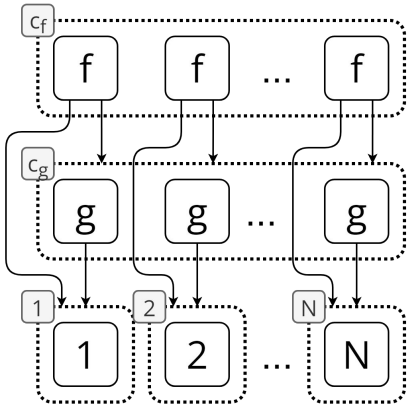


E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph

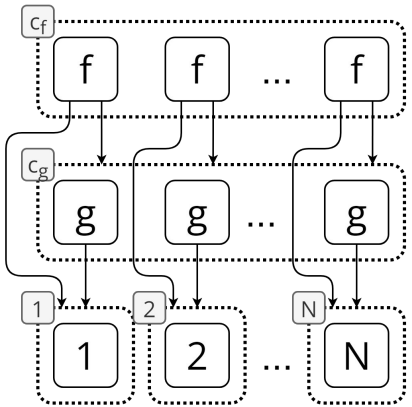
E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph



E-matching: pattern matching over e-graphs

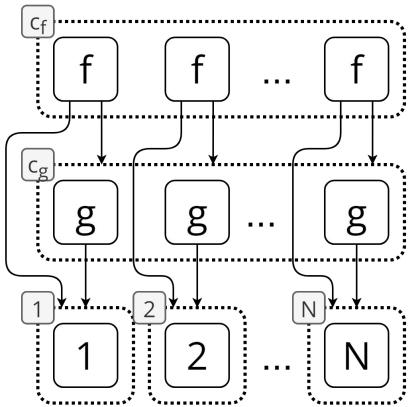
- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph



f(a, g(a)) will match *f(1, g(1))* *f(2, g(2))* ... *f(N, g(N))*, witnessed by *{a ↦ 1}* *{a ↦ 2}* ... *{a ↦ N}*.

E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph

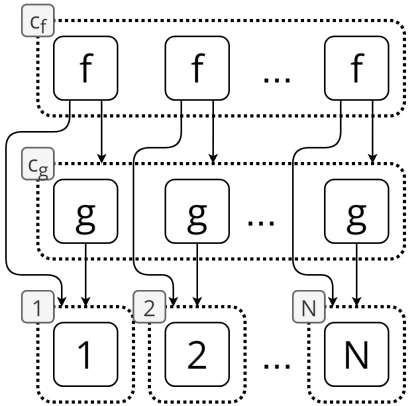


f(a, g(a)) will match *f(1, g(1))*, *f(2, g(2))*, ..., *f(N, g(N))*, witnessed by $\{a \mapsto 1\}$, $\{a \mapsto 2\}$, ..., $\{a \mapsto N\}$.

f(1, a) will match *f(1, g(1))*, *f(1, g(2))*, ..., *f(1, g(N))*, witnessed by $\{a \mapsto c_g\}$.

E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph
- NP-complete wrt to pattern size (Kozen 1977) 😱

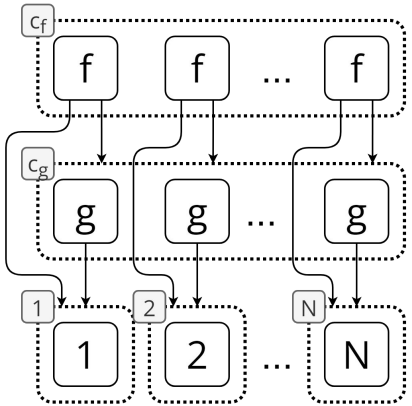


f(a, g(a)) will match *f(1, g(1))*, *f(2, g(2))*, ..., *f(N, g(N))*, witnessed by *{a ↦ 1}*, *{a ↦ 2}*, ..., *{a ↦ N}*.

f(1, a) will match *f(1, g(1))*, *f(1, g(2))*, ..., *f(1, g(N))*, witnessed by *{a ↦ c_g}*.

E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph
- NP-complete wrt to pattern size (Kozen 1977) 😱

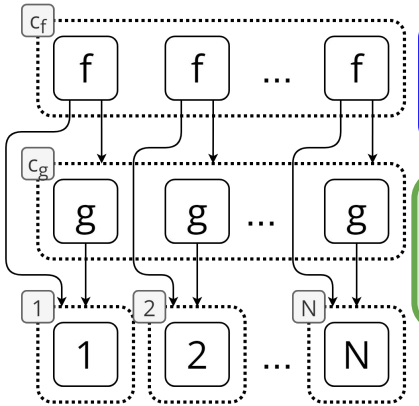


But patterns are often small…

f(1, α) will match *f(1, g(1))*, *f(1, g(2))*, ..., *f(1, g(N))*, witnessed by $\{\alpha \mapsto c_g\}$.

E-matching: pattern matching over e-graphs

- *E-matching* : find substs from pattern variables to e-classes
- Substs guaranteed to be represented by the matched e-graph
- NP-complete wrt to pattern size (Kozen 1977) 😱



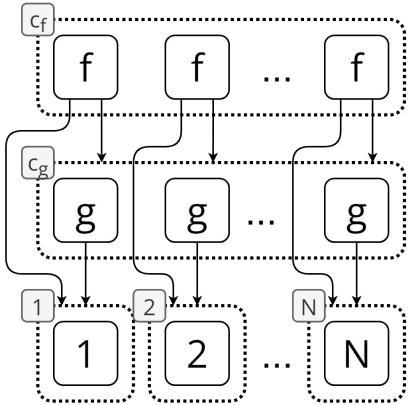
But patterns are often small…

💡 # of matches is better metric!

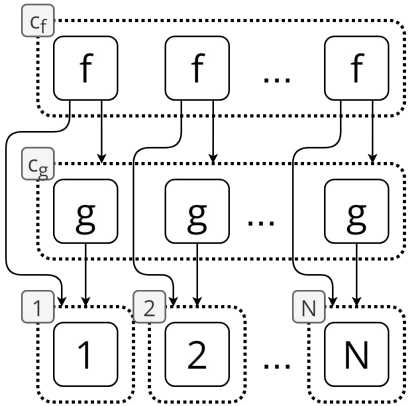
...

f(1, g(N))

Traditional e-matching via backtracking



Traditional e-matching via backtracking

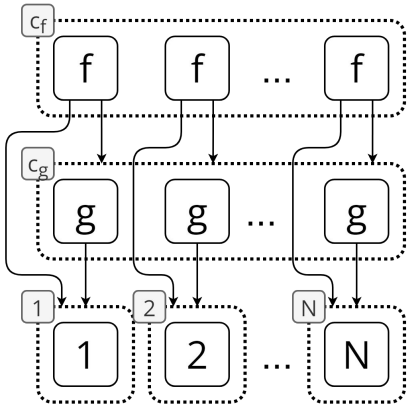


f(a, g(a))

```
for e-class c in e-graph E:
```

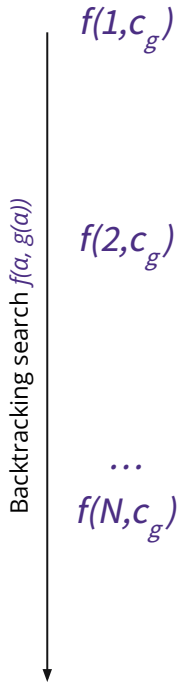
Backtracking search *f(a, g(a))*

Traditional e-matching via backtracking

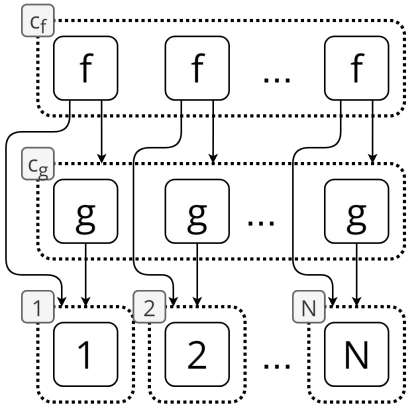


f(a, g(a))

```
for e-class c in e-graph E:
  for f-node n1 in c:
```



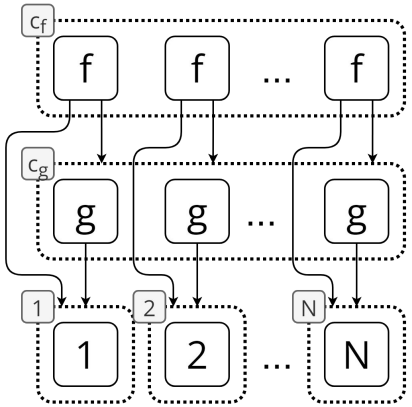
Traditional e-matching via backtracking



f(α, g(α))

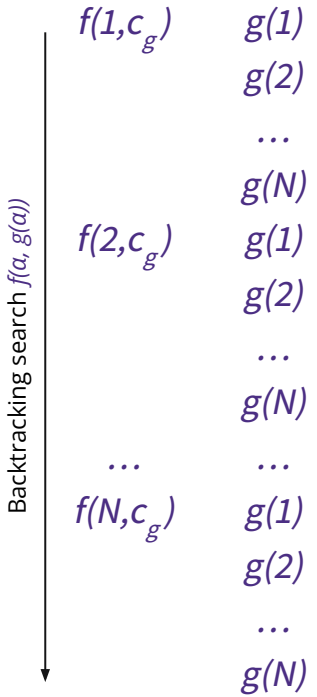
```
for e-class c in e-graph E:
  for f-node n1 in c:
    subst = {root ↦ c, α ↦ n1.child1}
```

Traditional e-matching via backtracking

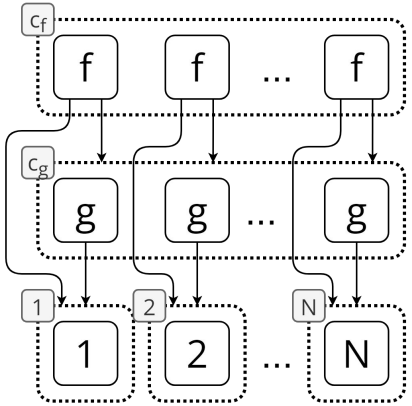


f(α, g(α))

```
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:
```



Traditional e-matching via backtracking



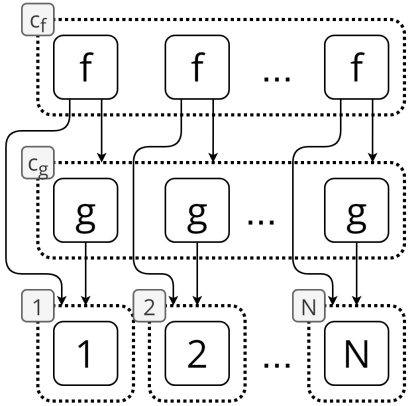
f(α, g(α))

```
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:
```

Backtracking search *f(α, g(α))*

$f(1,c_g)$	$g(1)$	✓
	$g(2)$	☐
	...	
	$g(N)$	☐
$f(2,c_g)$	$g(1)$	☐
	$g(2)$	✓
	...	
	$g(N)$	☐
...	...	
$f(N,c_g)$	$g(1)$	☐
	$g(2)$	☐
	...	
	$g(N)$	✓

Traditional e-matching via backtracking



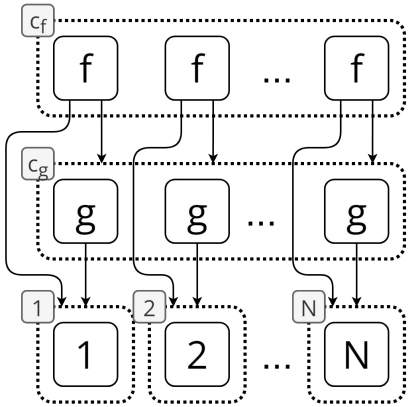
f(α, g(α))

```
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
```

Backtracking search *f(α, g(α))*

$f(1,c_g)$	$g(1)$	✓
	$g(2)$	☐
	...	
	$g(N)$	☐
$f(2,c_g)$	$g(1)$	☐
	$g(2)$	✓
	...	
	$g(N)$	☐
...	...	
$f(N,c_g)$	$g(1)$	☐
	$g(2)$	☐
	...	
	$g(N)$	✓

Traditional e-matching via backtracking



f(α, g(α))

```
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:
```

O(N^2), yet at most O(N) matches 😫

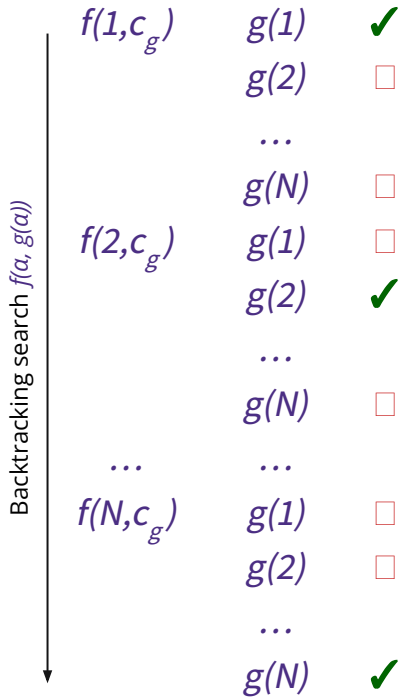
Backtracking search *f(α, g(α))*

$f(1,c_g)$	$g(1)$	✓
	$g(2)$	☐
	...	
	$g(N)$	☐
$f(2,c_g)$	$g(1)$	☐
	$g(2)$	✓
	...	
	$g(N)$	☐
...	...	
$f(N,c_g)$	$g(1)$	☐
	$g(2)$	☐
	...	
	$g(N)$	✓

Traditional e-matching via backtracking

- Many optimizations in literature
  - custom VMs for “CSE”
  - specific patterns
  - mod-time analysis

- No data complexity bounds!



Key insight: e-matching is a DB problem!

E-matching in e-graphs

Finding substitutions such that substituted terms are represented in an e-graph.

Conjunctive queries in DBs

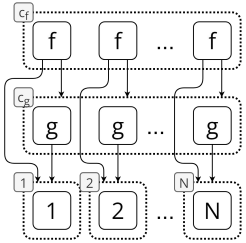
Finding substitutions such that substituted atoms are present in a relational DB.

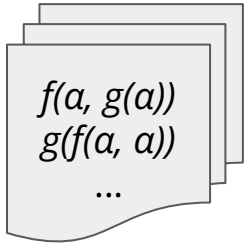
[POPL 2022]

Relational e-matching

Relational e-matching

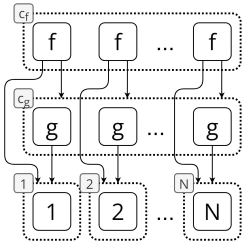
- Given e-graph + patterns

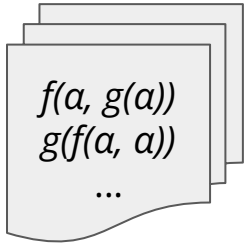




Relational e-matching

- Given e-graph + patterns
- Transform e-graph to tables





R_f

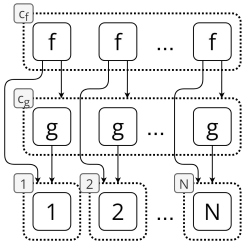
arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
...	...	...
N	c_g	c_f

R_g

arg_1	id
1	c_g
2	c_g
...	...
N	c_g

Relational e-matching

- Given e-graph + patterns
- Transform e-graph to tables
- Compile patterns to queries



R_f

arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
...	...	...
N	c_g	c_f

R_g

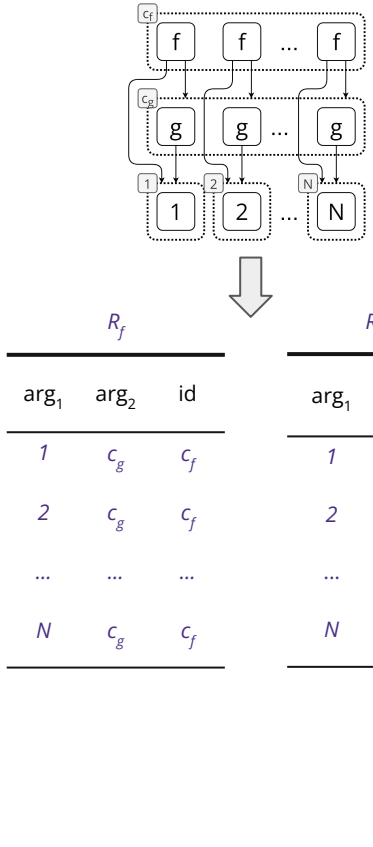
arg_1	id
1	c_g
2	c_g
...	...
N	c_g

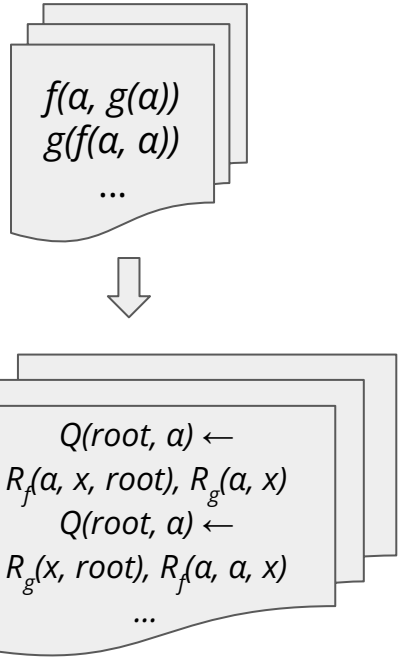
f(a, g(a))
g(f(a, a))
...

Q(root, a) ←
R_f(a, x, root), R_g(a, x)
Q(root, a) ←
R_g(x, root), R_f(a, a, x)
...

Relational e-matching

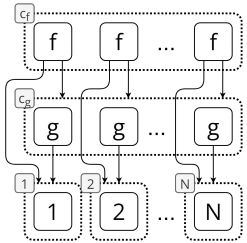
- Given e-graph + patterns
- Transform e-graph to tables
- Compile patterns to queries
- Use DB query engine to e-match!





Relational e-matching

- Given e-graph + patterns
- Transform e-graph to tables
- Compile patterns to queries
- Use DB query engine to e-match!
- Derive bounds from DB theory!



R_f

arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
...	...	...
N	c_g	c_f

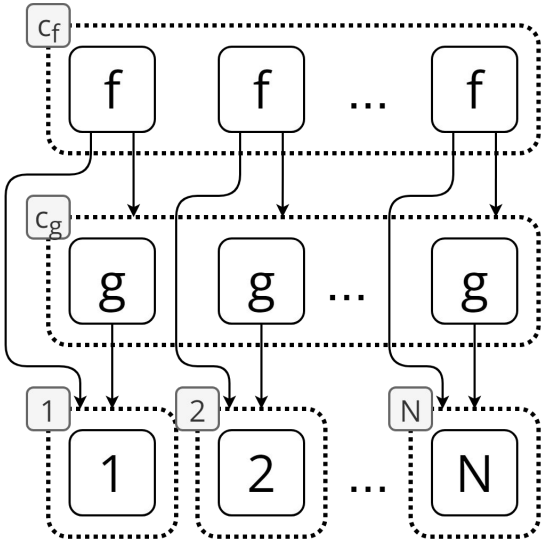
R_g

arg_1	id
1	c_g
2	c_g
...	...
N	c_g

f(a, g(a))
g(f(a, a))
...

Q(root, a) ←
R_f(a, x, root), R_g(a, x)
Q(root, a) ←
R_g(x, root), R_f(a, a, x)
...

E-graphs as tables (relational DBs)



R_f

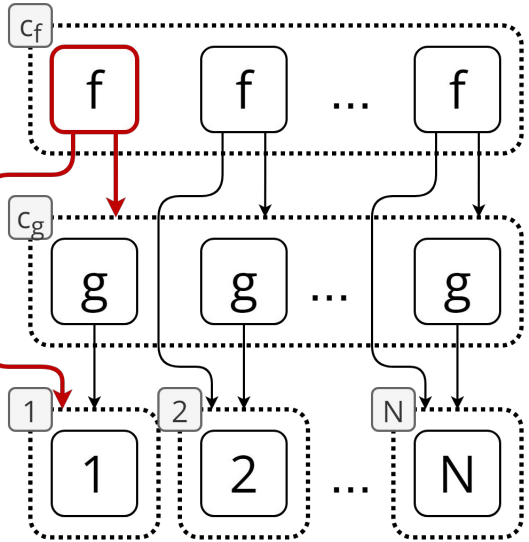
arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
...	...	...
N	c_g	c_f

R_g

arg_1	id
1	c_g
2	c_g
...	...
N	c_g

id	$R_{i=1...N}$
i	

E-graphs as tables (relational DBs)



each e-node becomes a row

R_f

arg_1	arg_2	id
1	c_g	c_f
2	c_g	c_f
...	...	...
N	c_g	c_f

R_g

arg_1	id
1	c_g
2	c_g
...	...
N	c_g

id
i

$R_{i=1...N}$

E-match patterns as conjunctive queries

$f(\alpha, g(\alpha))$

E-match patterns as conjunctive queries

$f(\alpha, g(\alpha))$

$Q(root, \alpha) \leftarrow$

$R_f(\alpha, x, root),$

$R_g(\alpha, x)$

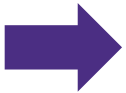
E-match patterns as conjunctive queries

$f(\alpha, g(\alpha))$

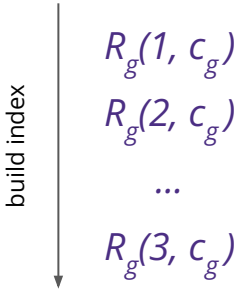
$Q(root, \alpha) \leftarrow$

$R_f(\alpha, x, root),$

$R_g(\alpha, x)$



```
ind = {}
for (α, x) in R_g: # build index
  ind.insert((α, x))
```



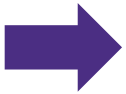
E-match patterns as conjunctive queries

$f(\alpha, g(\alpha))$

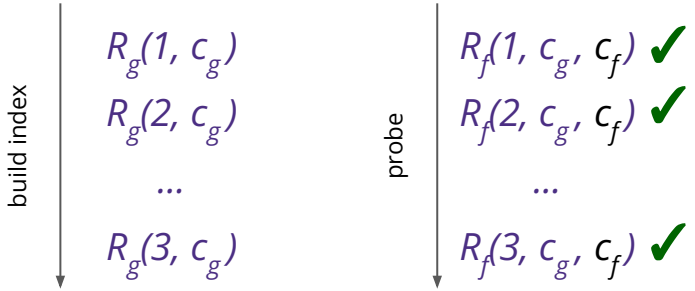
$Q(root, \alpha) \leftarrow$

$R_f(\alpha, x, root),$

$R_g(\alpha, x)$



```
ind = {}
for (α, x) in R_g: # build index
  ind.insert((α, x))
for (α, x, root) in R_f: # probe
  if (α, x) in ind:
    yield {root ↦ root, α ↦ α}
```



Why is relational e-matching faster?

$f(\alpha, g(\alpha))$

Enum all terms of shape $f(\alpha, g(\beta))$

Check if $\alpha = \beta$ only before yielding

$Q(root, \alpha) \leftarrow$

$R_f(root, \alpha, x), R_g(x, \alpha)$

Build indices on both α and x.

Only enum terms where constraints on both x and α are satisfied.

structural constraints

equality constraints

Data complexity results (see paper)

THEOREM 9. *Relational e-matching is worst-case optimal; that is, fix a pattern p, let $M(p, E)$ be the set of substitutions yielded by e-matching on an e-graph E with N e-nodes, relational e-matching runs in time $O(\max_E(|M(p, E)|))$.*

THEOREM 10. *Fix an e-graph E with N e-nodes that compiles to a database I, and a fix pattern p that compiles to conjunctive query $Q(\overline{X}) \leftarrow R_1(\overline{X_1}), \ldots, R_m(\overline{X_m})$. Relational e-matching p on E runs in time $O\left(\sqrt{|Q(I)| \times \Pi_i |R_i|}\right) \leq O\left(\sqrt{|Q(I)| \times N^m}\right)$.*

New Capabilities: *Multi-patterns*

```
(rule (
   (eq X (matmul a b))
   (eq Y (matmul a c))
)(
   (eq F (matmul a (concat b c))
   (eq X (split1 F))
   (eq Y (split2 F))
))
```

New Capabilities: *Multi-patterns*

```
(rule (
   (eq X (matmul a b))
   (eq Y (matmul a c))
)(
   (eq F (matmul a (concat b c))
   (eq X (split1 F))
   (eq Y (split2 F))
))
```

search for multiple patterns *anywhere* in the e-graph

New Capabilities: *Multi-patterns*

```
(rule (
   (eq X (matmul a b))
   (eq Y (matmul a c))
)(
   (eq F (matmul a (concat b c))
   (eq X (split1 F))
   (eq Y (split2 F))
))
```

search for multiple patterns *anywhere* in the e-graph

perform multiple merges, each on separate e-classes!

New Capabilities: *Merge-only rules*

```
(rule (
    (eq R (+ (+ a b) c))
)(
    (eq R (+ a (+ b c)))
))
```

New Capabilities: *Merge-only rules*

```
(rule (
    (eq R (+ (+ a b) c))
)(
    (eq R (+ a (+ b c)))
))
```

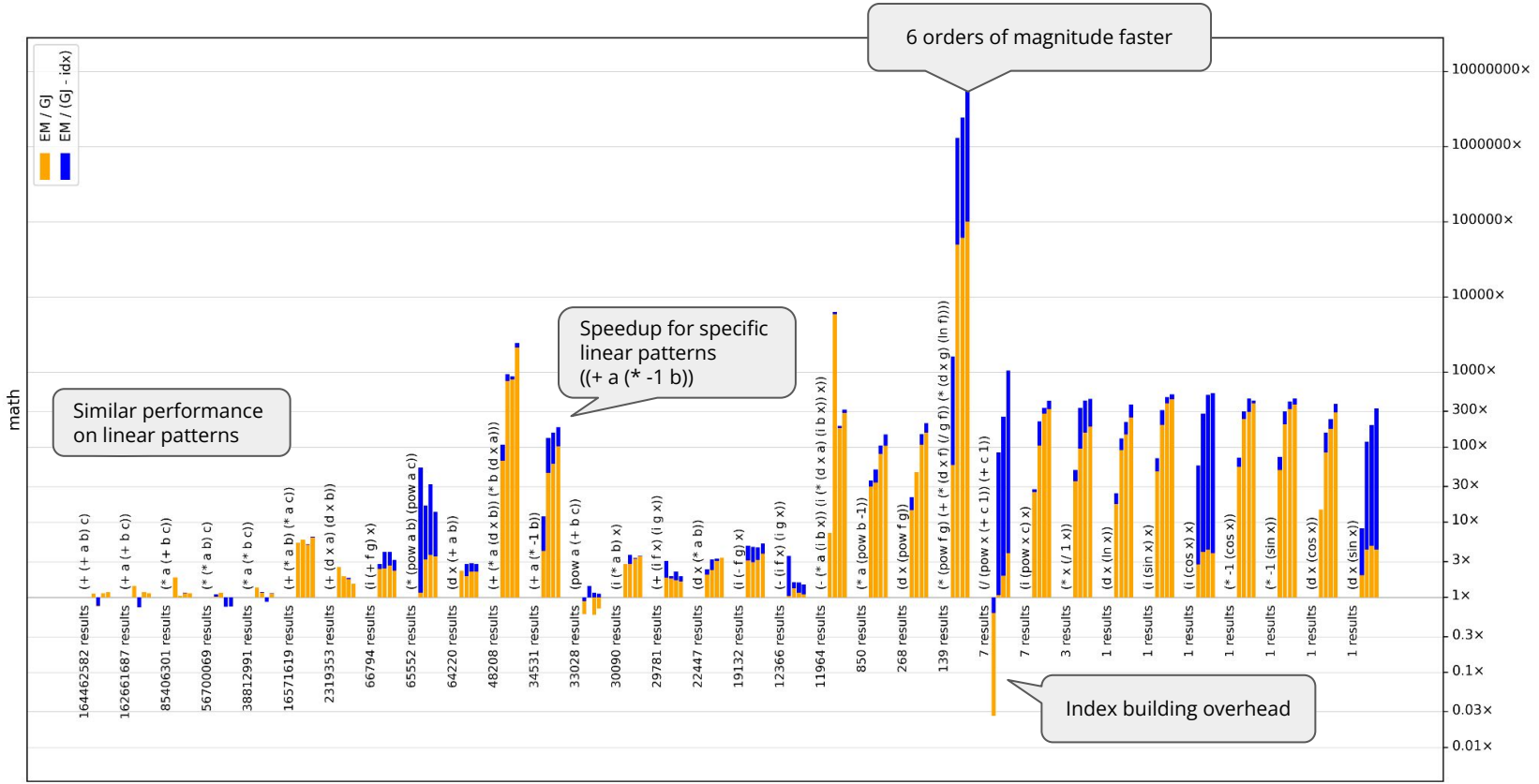
versus

```
(rule (
    (eq R1 (+ (+ a b) c))
    (eq R2 (+ a (+ b c)))
)(
    (eq R1 R2)
))
```

Will never add a new e-class.

Can only “shrink” the e-graph!

Relational e-matching : asymptotic speedup



Graph → Tables → Graph → Tables → …

- Relational e-match prototype: repeated graph ⇔ DB conversion
- **egglog** : fully embrace DB
  - Now EqSat “just” Datalog + congruence!
  - Need to keep fast invariant maintenance
  - Want to leverage Datalog strengths as well

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* amortizes maintaining this invariant in DB!

For terms, congruence violation manifests as “same args, diff class”

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* amortizes maintaining this invariant in DB!

For terms, congruence violation manifests as “same args, diff class”

Foo

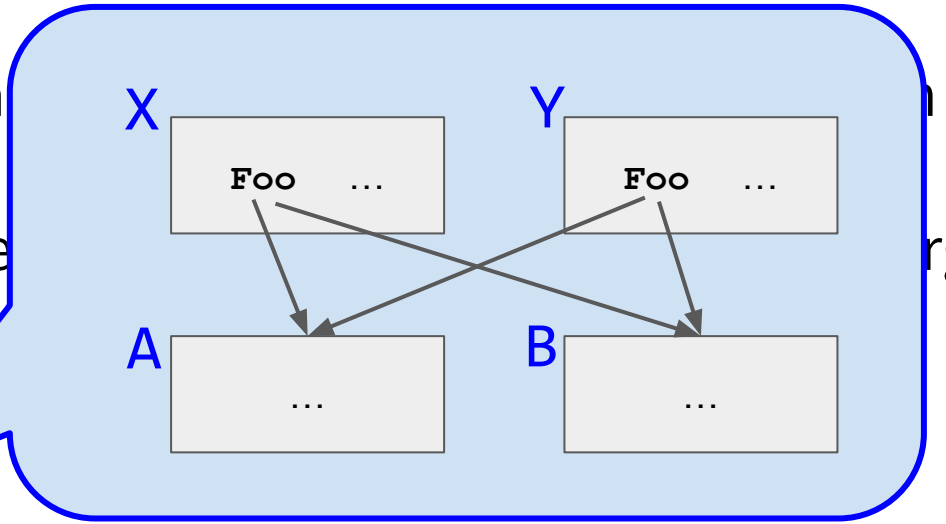
ArgL	ArgR	EC
A	B	X
A	B	Y

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* am … DB!

For terms, congrue … rgs, diff class”



Foo

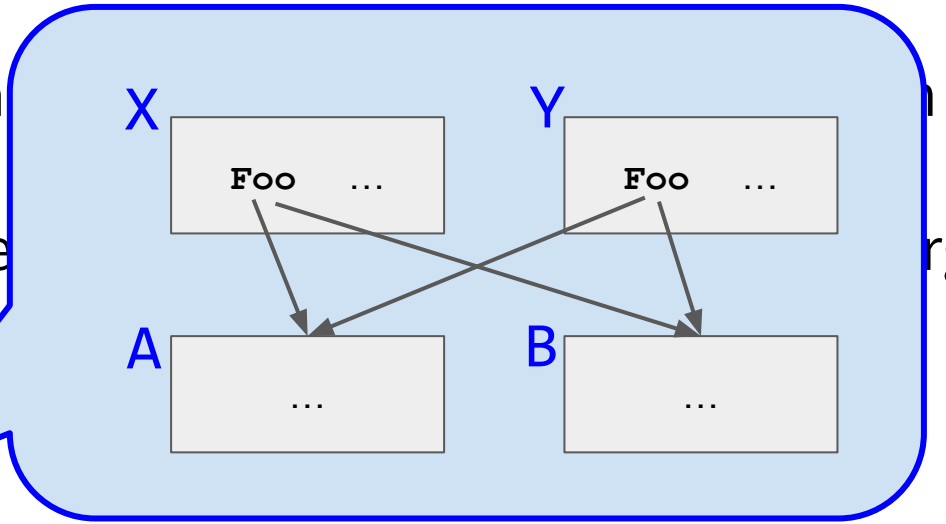
ArgL	ArgR	EC
A	B	X
A	B	Y

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* am … DB!

For terms, congrue … rgs, diff class”



Foo

ArgL	ArgR	EC
A	B	X
A	B	Y

In tables = FD violation!

Congruence: just a functional dependence!

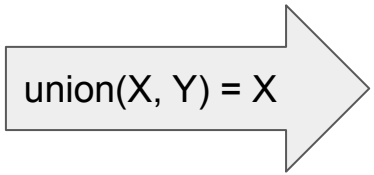
Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* amortizes maintaining this invariant in DB!

For terms, congruence violation manifests as “same args, diff class”

Foo

ArgL	ArgR	EC
A	B	X
A	B	Y



Foo

ArgL	ArgR	EC
A	B	X

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* amortizes maintaining this invariant in DB!

For terms, congruence violation manifests as “same args, diff class”

Foo

ArgL	ArgR	EC
A	B	X
A	B	Y

union(X, Y) = X

Foo

ArgL	ArgR	EC
A	B	X

May cause additional congruence merges!

Congruence: just a functional dependence!

Remember key e-graph invariant: **a = b ⇒ f(a) = f(b)**

Rebuilding *still* amortizes maintaining this invariant in DB!

For terms, congruence violation manifests as “same args, diff class”

Foo

ArgL	ArgR	EC
A	B	X
A	B	Y

union(X, Y) = X

Foo

ArgL	ArgR	EC
A	B	X

May cause additional congruence merges!

Merge in the union find and keep rebuilding to fixpoint!

New Capabilities: *Incremental e-matching*

Just Datalog semi-naive eval!

$$Q = R \bowtie S \bowtie T$$

$$\begin{aligned} \Delta Q = {} & \Delta R \bowtie S \bowtie T \\ & \cup\ R \bowtie \Delta S \bowtie T \\ & \cup\ R \bowtie S \bowtie \Delta T \end{aligned}$$

PL Problems

1. Phase Ordering 😭 ⇒ 🤩
2. Syntactic Brittleness 😭
3. …

Keep simple + modular!

But make robust?

DB techniques will come to our rescue!

PL Problems

1. Phase Ordering 😭 ⇒ 🤩
2. Syntactic Brittleness 😭
3. …

Keep simple + modular!

But make robust?

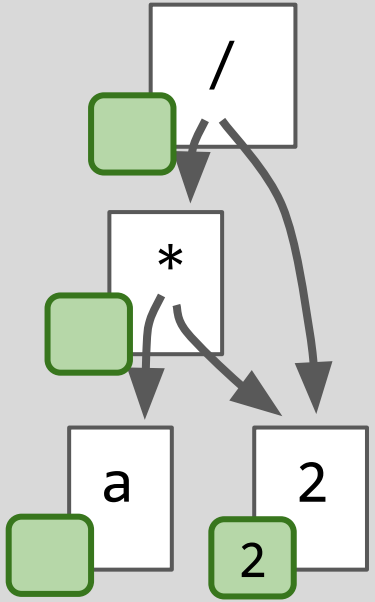
DB techniques will come to our rescue!

Syntactic rewriting is not enough…

- How many rules do we need for constant folding?
  - `2 + 2 → 4,  3 + 4 → 7,  4 + 6 → 10, …` *a lot!*
- What about satisfying guards for conditional rules?
  - `x / x → 1  only ok if  x ≠ 0`
- In general, many optimizations depend on analyses!
  - `nullability, tensor shape, intervals, free variables, …`

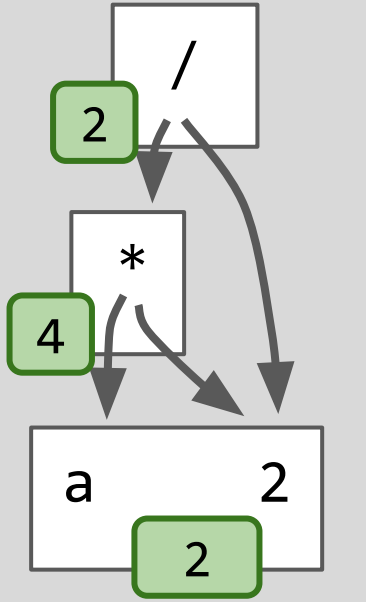
Constant folding

- Option<Number> “fact” per e-class
- Try to eval new e-nodes
- Option “or” on merge



Constant folding

- Option<Number> “fact” per e-class
- Try to eval new e-nodes
- Option “or” on merge
- Only propagates up!



Original egg *e-class analyses* too limited:

- Facts could only propagate from children to parents
  - Injectivity difficult for `f` where `f(x) = f(y) → x = y`
- Analyses / rules implemented + executed separately
  - Only “one kind” of fact per domain (products clunky)
  - Custom Rust code to realize conditional rewrites
  - Difficult to give unified theory of EqSat! Termination?

Program analysis modulo equivalence

- Tightest summary over all equivalent represented terms!

To demonstrate an advantage of this approach, consider the following example, for $x \in [0, 1], y \in [1, 2]$, where the following concrete-equivalences are discovered via rewriting:

$$1 - \frac{2y}{x+y} \in \left[-3, \frac{1}{3}\right] \quad (5)$$

$$\cong \frac{x-y}{x+y} \in [-2, 0] \quad (6)$$

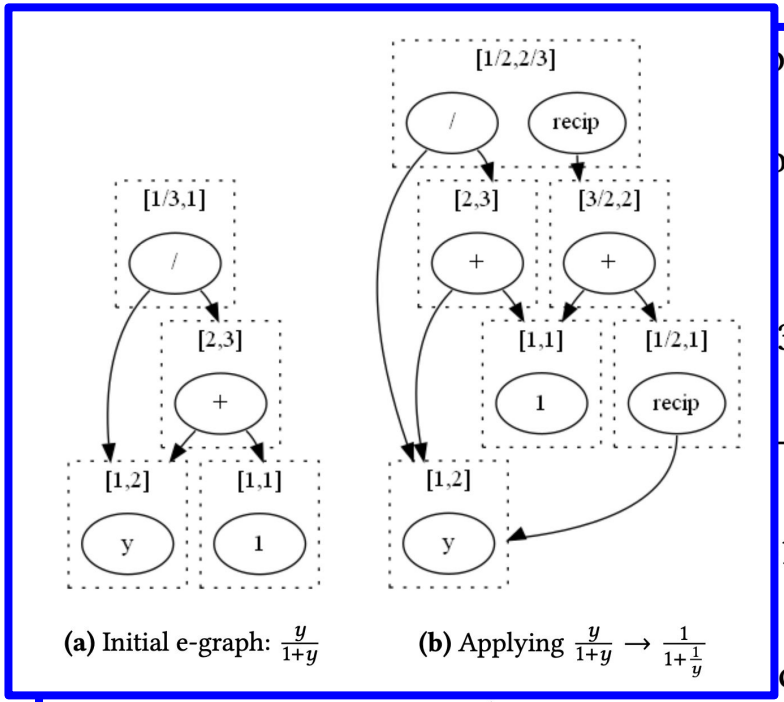
$$\cong \frac{2x}{x+y} - 1 \in [-1, 1]. \quad (7)$$

The interval associated with the e-class containing these three expressions is $[-3, \frac{1}{3}] \cap [-2, 0] \cap [-1, 1] = [-1, 0]$. We

Sam Coward et al. (2022)

Program analysis modulo equivalence

- Tightest summary over all equivalent represented terms!
- *Virtuous cycle:* facts enable rewrites, rewrites improve facts!



pproach, consider
[1, 2], where the
overed via rewrit-

$$3, \frac{1}{3}\Big] \quad (5)$$

$$-2, 0] \quad (6)$$

$$1, 1]. \quad (7)$$

containing these

three expressions is $[-3, \frac{1}{3}] \cap [-2, 0] \cap [-1, 1] = [-1, 0]$. We

Sam Coward et al. (2022)

egglog: Analyses via FDs and “merge functions”

- **Key Idea** : table for function F has args → e-class FD

egglog: Analyses via FDs and “merge functions”

- **Key Idea** : table for function F has args → e-class FD
- For “terms” enforcing FD should restore congruence

egglog: Analyses via FDs and “merge functions”

- **Key Idea** : table for function F has args → e-class FD
- For “terms” enforcing FD should restore congruence ✅
- But what about analysis facts? Can’t just union…

egglog: Analyses via FDs and “merge functions”

- **Key Idea** : table for function F has args → e-class FD
- For “terms” enforcing FD should restore congruence ✅
- But what about analysis facts? Can’t just union…
  - Facts are also just functions!
  - **Allow user-specified FD repair!**

egglog: FDs and “merge functions”

```
(datatype Math
  (Num Rational)
  (Var String)
  (Const String)
  (Add Math Math)
  (Sub Math Math)
  (Mul Math Math)
  (Div Math Math)
  (Neg Math)
  (Sqrt Math)
  …)
```

egglog: FDs and “merge functions”

```
(datatype Math
  (Num Rational)
  (Var String)
  (Const String)
  (Add Math Math)
  (Sub Math Math)
  (Mul Math Math)
  (Div Math Math)
  (Neg Math)
  (Sqrt Math)
  …)
```

```
(function hi (Math) Rational :merge min)
(function lo (Math) Rational :merge max)
(relation non-zero (Math))
```

egglog: FDs and “merge functions”

```
(datatype Math
  (Num Rational)
  (Var String)
  (Const String)
  (Add Math Math)
  (Sub Math Math)
  (Mul Math Math)
  (Div Math Math)
  (Neg Math)
  (Sqrt Math)
  …)
```

```
(function hi (Math) Rational :merge min)
(function lo (Math) Rational :merge max)
(relation non-zero (Math))
```

```
(rewrite (Add (Num a) (Num b))
         (Num (+ a b)))
(rewrite (Sub (Num a) (Num b))
         (Num (- a b)))
(rewrite (Mul (Num a) (Num b))
         (Num (* a b)))
(rewrite (Div (Num a) denom)
         (Num (/ a b))
  :when ((= denom (Num b))
         (non-zero denom)))
…
```

egglog: FDs and “merge functions”

```
(datatype Math
  (Num Rational)
  (Var String)
  (Const String)
  (Add Math Math)
  (Sub Math Math)
  (Mul Math Math)
  (Div Math Math)
  (Neg Math)
  (Sqrt Math)
  …)
```

```
(function hi (Math) Rational :merge min)
(function lo (Math) Rational :merge max)
(relation non-zero (Math))
```

```
(rule ((= e (Add a b))
       (= la (lo a))
       (= lb (lo b)))
      ((set (lo e) (+ la lb))))

(rule ((= e (Add a b))
       (= ha (hi a))
       (= hb (hi b)))
      ((set (hi e) (+ ha hb))))
```

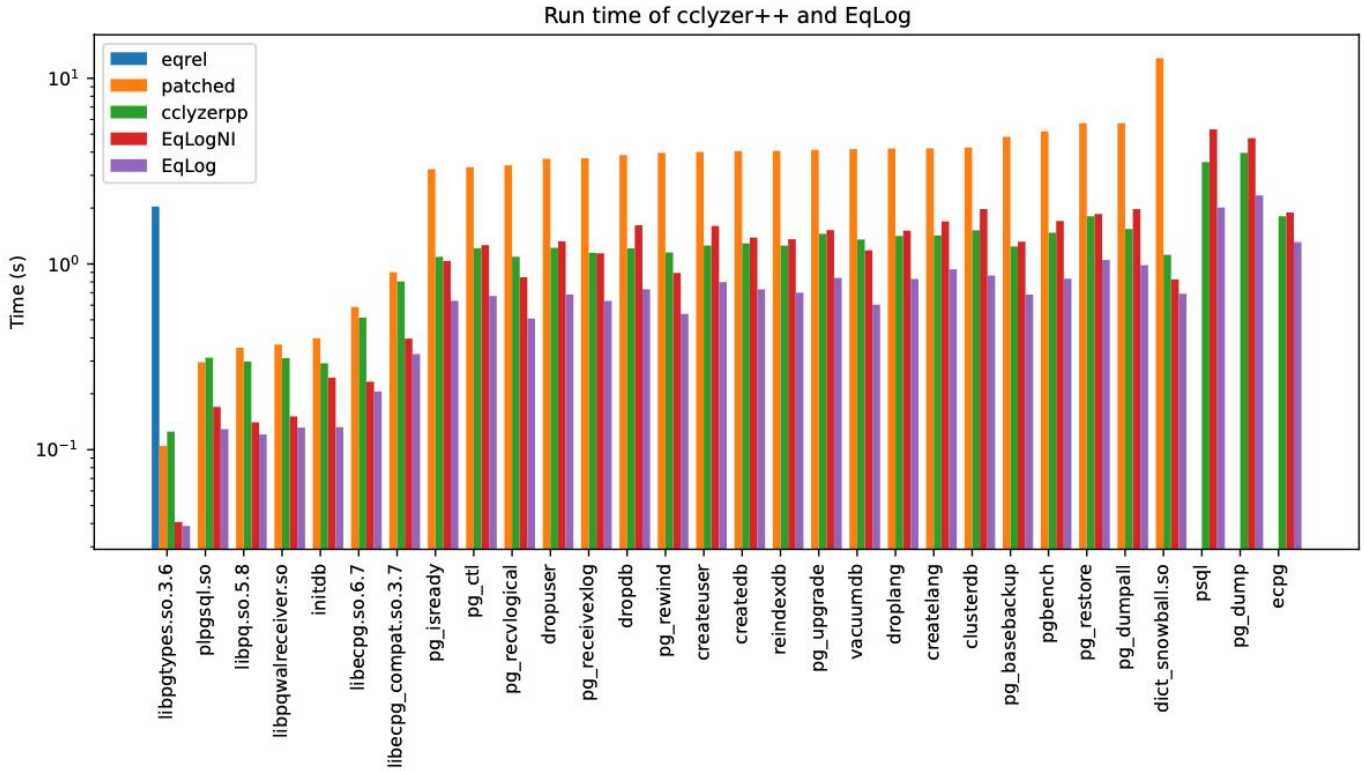
egglog: FDs and “merge functions”

```
(datatype Math
  (Num Rational)
  (Var String)
  (Const String)
  (Add Math Math)
  (Sub Math Math)
  (Mul Math Math)
  (Div Math Math)
  (Neg Math)
  (Sqrt Math)
  …)
```

```
(function hi (Math) Rational :merge min)
(function lo (Math) Rational :merge max)
(relation non-zero (Math))
```

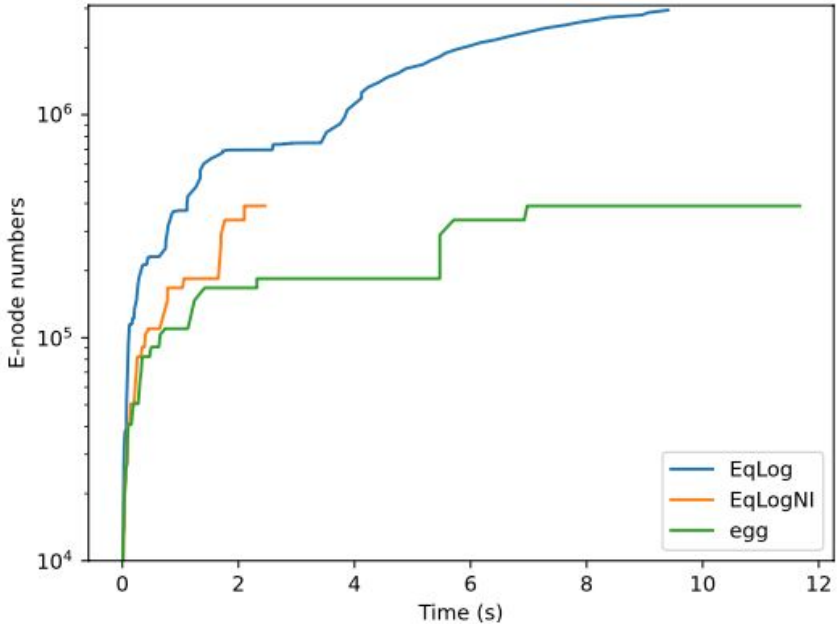
```
(rule ((= l (lo e))
       (> l r-zero))
      ((non-zero e)))

(rule ((= h (hi e))
       (< h r-zero))
      ((non-zero e)))

(rule ((= e (Num ve)))
      ((set (lo e) ve)
       (set (hi e) ve)))
```

egglog Pointer Analysis



Huge egglog Wins

- No graph ⇔ DB conversions
- Analyses now “just” rewrites
- Semi-naive evaluation → incremental e-matching!
- Provides easy text format
- Key: flexible framework!!
- Support unified EqSat/DB theory



PL Problems

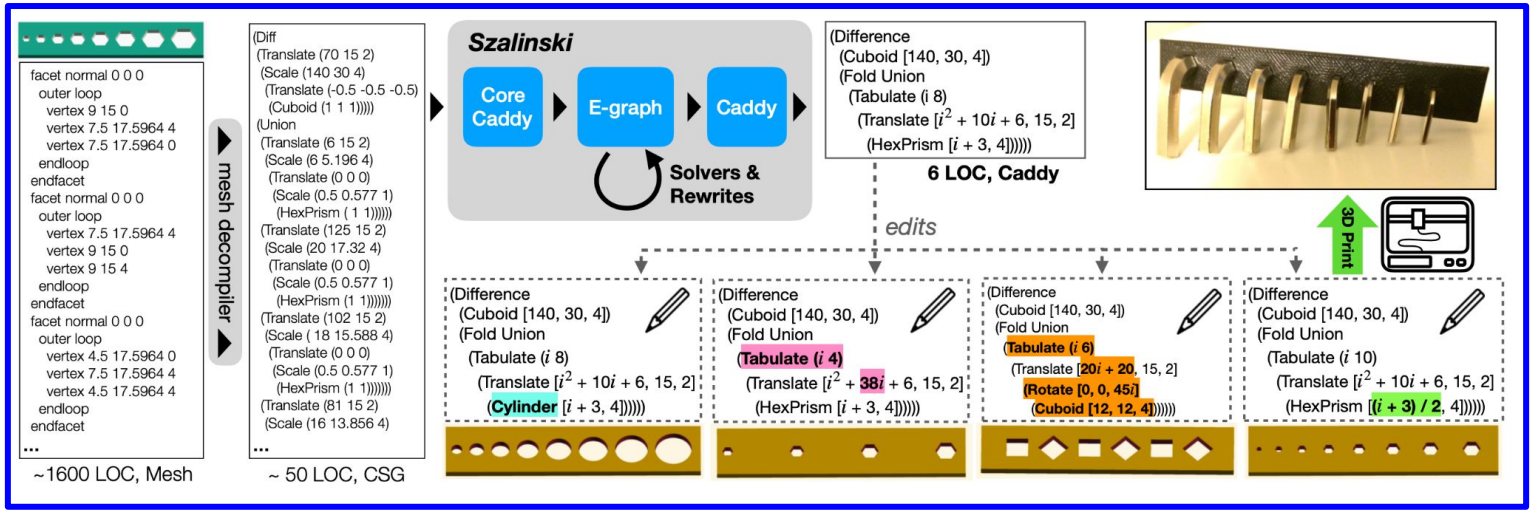
1. Phase Ordering 😭 ⇒ 🤩
2. Syntactic Brittleness 😭 ⇒ 🤩
3. …

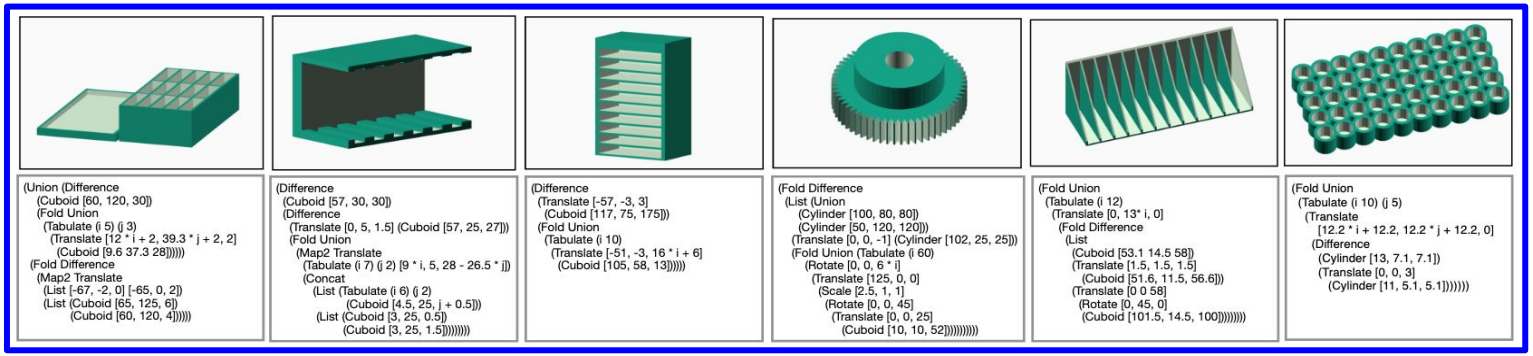
Keep simple + modular!

But make robust?

DB techniques will come to our rescue!

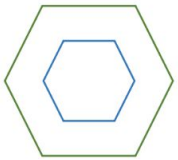
Decompile 3D CAD [ICFP 2018, PLDI 2020, POPL 2023]





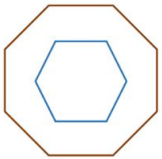
Decompile 3D CAD [ICFP 2018, PLDI 2020, POPL 2023]

A Initial corpus (size 208)



```
(combine
  (xform
    (repeat (xform line ...) 6 [1 (2π / 6) 0 0])
    [2 0 0 0])
  (xform
    (repeat (xform line ...) 6 [1 (2π / 6) 0 0])
    [1 0 0 0]))
```

```
(combine (combine
 (xform
    (repeat (xform line ...) 6 [1 (2π / 6) 0 0])
    [2.25 0 0 0]))
 (xform
    (repeat (xform line ...) 6 [1 (2π / 6) 0 0])
    [2 0 0 0]))
  (xform
    (repeat (xform line ...) 6 [1 (2π / 6) 0 0])
    [0.5 0 0 0]))
```



```
(combine
  (xform
    (repeat (xform line ...) 8 [1 (2π / 8) 0 0])
    [2 0 0 0])
  (xform
    (repeat (xfrom line ...) 6 [1 (2π / 6) 0 0])
    [1 0 0 0]))
```

library learning →

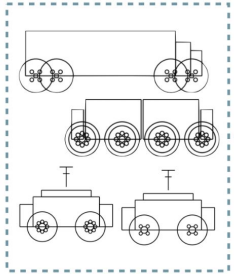
B Compressed corpus (size 72)

library:

```
f0 = λx0 x1 ->          // polygon with x1 sides scaled by x0
  xform
    (repeat (xform line ...) x1 [1 (2π / x1) 0 0])
    [x0 0 0 0]
```

refactored programs:

```
(combine
  (f0 6 2)
  (f0 6 1))

(combine (combine
  (f0 6 2.25)
  (f0 6 2))
  (f0 6 0.5))
```



```
oneseg_body = λrest →
  C (C (C (C (T (T (r_s 0 0) (xform_x 0)) (xform_x -8))
           (T (T (r_s 16 4.5) (M 1 0 0 2.25))
              (xform_x 0)))
        (T (T (r_s 0 0) (xform_x 0)) (xform_x 8)))
     (T (r_s 12 1) (M 1 0 0 5))) rest
```

oneseg_body <antenna> | oneseg_body <wheels>

one-segment vehicle body

```
circle_ring = λn → repeat
 (T (T c (M 0.25 0 0 0))
    (M 1 0 0.53033 0.53033))
n
(M 1 ((2 * π) / x0) 0 0)
```

circle_ring 4 | circle_ring 8

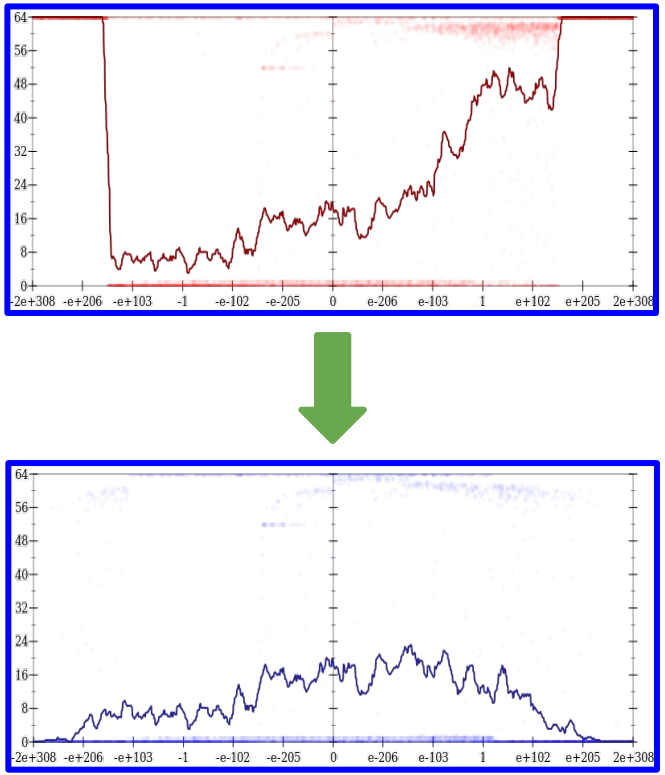
ring of circles

Compile $\mathbb{R}$ to IEEE 754 [PLDI 2015, ARITH 2021, ASPLOS 2025]

$$\frac{(-b) + \sqrt{b \cdot b - 4 \cdot (a \cdot c)}}{2 \cdot a}$$

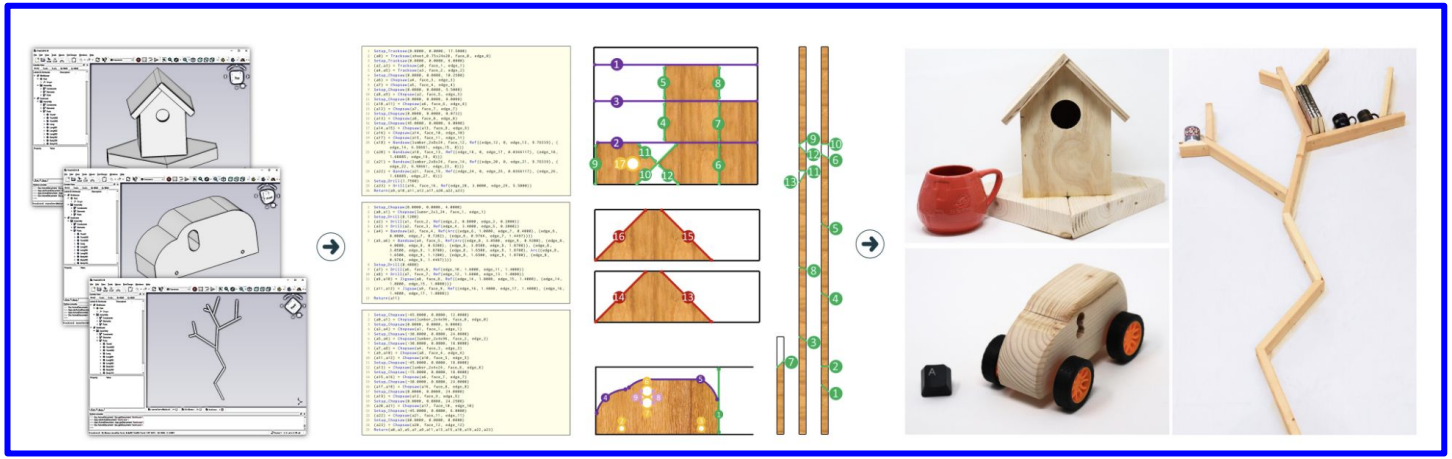
$$\downarrow$$

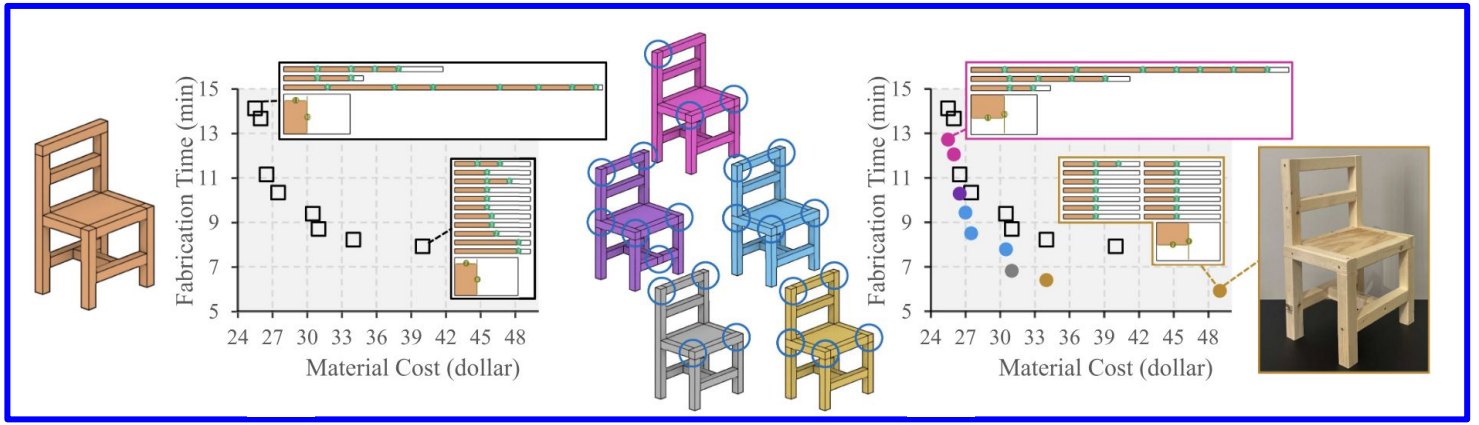
$$\begin{array}{l}
\textbf{if } b \leq -2.1714197031320663 \cdot 10^{+114} : \\
\quad -\frac{b}{a} \\
\textbf{elif } b \leq 2.9809086538561536 \cdot 10^{-153} : \\
\quad \frac{\sqrt{\mathsf{fma}(b,b,c \cdot (a \cdot -4))} - b}{a \cdot 2} \\
\textbf{elif } b \leq 3.095118518558678 \cdot 10^{+20} : \\
\quad t_0 := 4 \cdot (a \cdot c) \\
\quad \frac{t_0 \cdot \frac{0.5}{a}}{(-b) - \sqrt{b \cdot b - t_0}} \\
\textbf{else} : \\
\quad -\frac{c}{b}
\end{array}$$



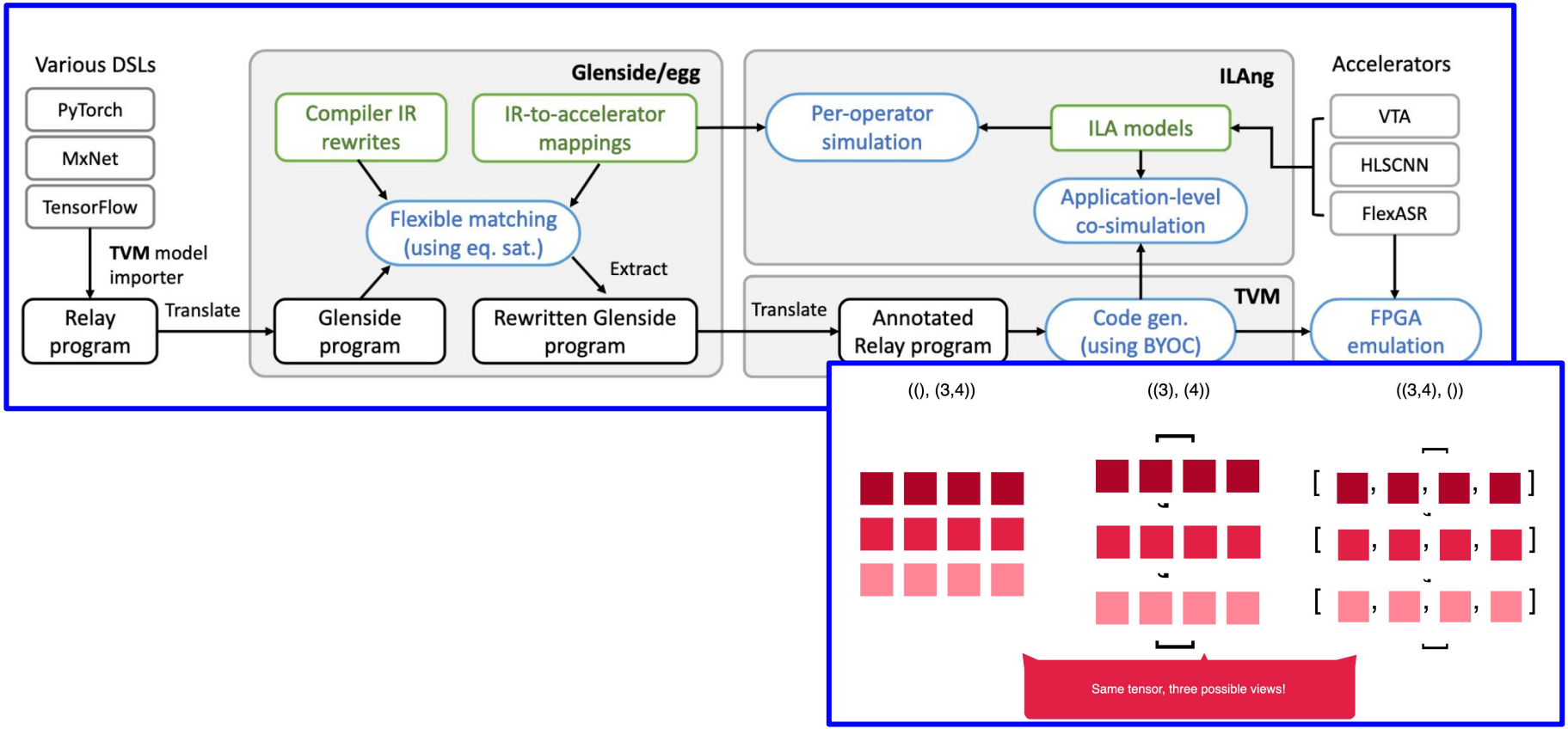
HERBIE

Carpentry Compiler [SIGGRAPH ASIA 2019, TOG 2022]





Targeting HW Accelerators [MAPS 2021, TODAES 2023]



Many Other Applications!

Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors

Samuel Thomas
sgt@cs.utexas.edu
The University of Texas at Austin, TX, USA

Vectorization for Digital Signal Processors via Equality Saturation

Alexa VanHattum
Rachit Nigam
Cornell University
Ithaca, NY, USA
Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA
Adrian Sampson
Cornell University

SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra

Yisu Remy Wang
University of Washington
Seattle, Washington
remywang@cs.washington.edu

Shana Hutchison
University of Washington
Seattle, Washington
shutchis@cs.washington.edu

Jonathan Lean
University of Washi
Seattle, Washing
jleang@cs.washing

Bill Howe
University of Washington
Seattle, Washington
billhowe@cs.washington.edu

Dan Suciu
University of Washington
Seattle, Washington
suciu@cs.washington.edu

Automating Constraint-Aware Datapath Optimization using E-Graphs

Samuel Coward
Numerical Hardware Group
Intel Corporation
Email: samuel.coward@intel.com

George A. Constantinides
Electrical and Electronic Engineering
Imperial College London
Email: g.constantinides@imperial.ac.uk

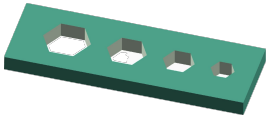
Theo Drane
Numerical Hardware Group
Intel Corporation
Email: theo.drane@intel.com


OFFLINE

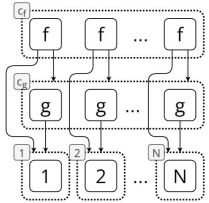
Thank You!

egraphs-good.github.io • egraphs.org

1. Rule-based transformations everywhere



2. E-graphs and Equality Saturation (EqSat)



3. Adapting DB results: more speed and flexibility

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

PLSE