Compiler Optimizations with Datalog and Equality Saturation

Anjali Pal

PLSE

```
def foo(x):
  return x + 1


print foo(1)
```

```
def foo(x):
  return x + 1


print foo(1)
```

```
print 2
```

```
if x < 2:
  y = x + x
  r = y * 2
else:
  r = x + x

return r
```

```
if x < 2:
  y = x + x
  r = y * 2
else:
  r = x + x

return r
```

```
r = x + x
if x < 2:
  r = r * 2
return r
```

```python
if x == 4:
  y = 5 * x
elif x == 5:
  y = 4 * x
else:
  y = 20
return y
```

```
if x == 4:
  y = 5 * x
elif x == 5:
  y = 4 * x
else:
  y = 20
return y
```

```
return 20
```

```
if x > 0:
  y = x
else:
  y = -x
z = y >= 0
return z
```

```
if x > 0:
  y = x
else:
  y = -x
z = y ≥ 0
return z
```

```
return true
```

```
if x > 0:
  y = 2
else:
  y = 3
if y < 10:
  z = 5
else:
  z = -5
return z
```

```
if x > 0:
  y = 2
else:
  y = 3
if y < 10:
  z = 5
else:
  z = -5
return z
```

```
return 5
```

Optimizations via rewrite rules in eggcc, our new compiler built with *egglog*

egglog: a fixpoint reasoning system
that combines *Datalog* and *Equality Saturation*

Optimizations via rewrite rules in eggcc,
our new compiler built with *egglog*

Datalog

Equality Saturation

egglog: a fixpoint reasoning system
that combines *Datalog* and *Equality Saturation*

Optimizations via rewrite rules in eggcc,
our new compiler built with *egglog*

Datalog

Equality Saturation

egglog: a fixpoint reasoning system
that combines *Datalog* and *Equality Saturation*

Optimizations via rewrite rules in **eggcc**,
our new compiler built with *egglog*

Datalog

Datalog

Datalog is a declarative programming language

consisting of **facts** and **rules**

Datalog

Datalog is a declarative programming language

consisting of **facts** and **rules**

Datalog Fact

```
Parent("Alice", "Bob")
```

Datalog Fact

```
Parent(“Alice”, “Bob”)
```

Alice is a parent of Bob

Datalog Rule

```
Ancestor(X, Y) :-
    Parent(X, Y)
```

Datalog Rule

```
Ancestor(X, Y) :-
    Parent(X, Y)
```

If X is a parent of Y,
then X is an ancestor of Y

Datalog Rule

```
Ancestor(X, Z) :-
    Parent(X, Y),
    Ancestor (Y, Z)
```

Datalog Rule

```
Ancestor(X, Z) :-
    Parent(X, Y),
    Ancestor (Y, Z)
```

If X is a parent of Y
and Y is an ancestor of Z,
then X is an ancestor of Z

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

What does the program mean?

How do we evaluate it?

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent

Ancestor

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent("Alice", "Bob")
Parent("Bob, "Charlie")
```

Parent	
Alice	Bob

Ancestor

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent	
Alice	Bob
Bob	Charlie

Ancestor

Datalog > EqSat > egglog > eggcc

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent

Alice	Bob
Bob	Charlie

Ancestor

Alice	Bob

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent	
Alice	Bob
Bob	Charlie

Ancestor	
Alice	Bob
Bob	Charlie

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent	
Alice	Bob
Bob	Charlie

Ancestor	
Alice	Bob
Bob	Charlie
Alice	Charlie

Datalog Program: Example

Program

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

Parent	
Alice	Bob
Bob	Charlie

Apply rules until there are no more facts to add

“fixpoint”

Alice	Bob
Bob	Charlie
Alice	Charlie

Datalog: Summary

- Program consists of **facts** and **rules**
- Rules can match on complex queries relating multiple facts
- Evaluate program by iteratively applying rules until fixpoint

Equality Saturation

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

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

Rewrite Rules

```
(x * y) / z ⟹ x * (y / z)        x * 2 ⟹ x << 1
 x / x      ⟹ 1                  x * y ⟹ y * x
 x * 1      ⟹ x                  x     ⟹ x * 1
```

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

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

Rewrite Rules

```
(x * y) / z ⟹ x * (y / z)        x * 2 ⟹ x << 1
 x / x      ⟹ 1                  x * y ⟹ y * x
 x * 1      ⟹ x                  x     ⟹ x * 1
```

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

$$(a * 2) / 2 \Longrightarrow (a \ll 1) / 2$$

Rewrite Rules

```
(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

(a * 2) / 2 ⟹ (2 * a) / 2 ⟹ (a * 2) / 2 ⟹ ...

Rewrite Rules

```
(x * y) / z ⟹ x * (y / z)        x * 2 ⟹ x << 1
 x / x      ⟹ 1                  x * y ⟹ y * x
 x * 1      ⟹ x                  x     ⟹ x * 1
```

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

$$a \Longrightarrow a * 1 \Longrightarrow (a * 1) * 1 \Longrightarrow \ldots$$

Rewrite Rules

```
(x * y) / z ⟹ x * (y / z)        x * 2 ⟹ x << 1
 x / x      ⟹ 1                  x * y ⟹ y * x
 x * 1      ⟹ x                  x     ⟹ x * 1
```

Datalog | EqSat | egglog | eggcc

(a * 2) / 2 ⟹ a

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 ⇒ a

But critical for other inputs!

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 → a

Which rewrite? When?

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
```

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

All of them at once!

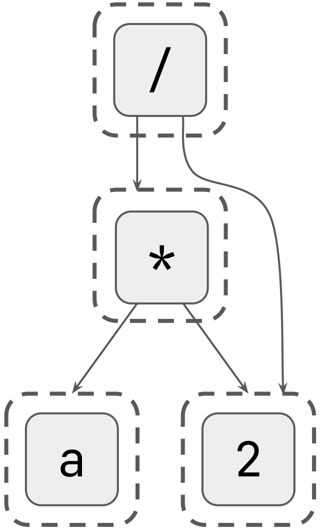
USEFUL

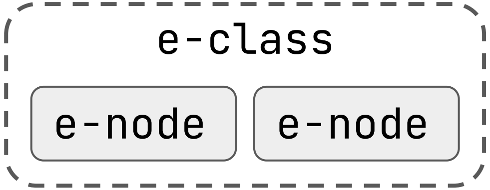
```
(x * y) / z ⟹ x * (y / z)
 x / x      ⟹ 1
 x * 1      ⟹ x
```

NOT SO USEFUL

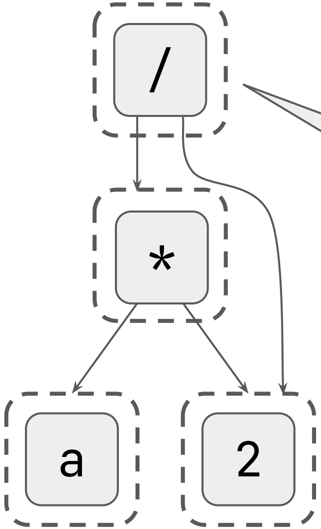
```
x * 2 ⟹ x << 1
x * y ⟹ y * x
x     ⟹ x * 1
```

Equivalence Graphs (e-graphs)



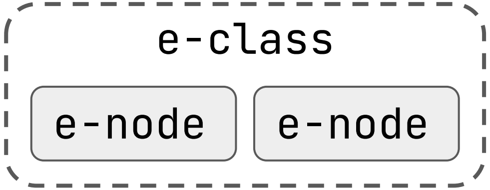


Equivalence Graphs (e-graphs)



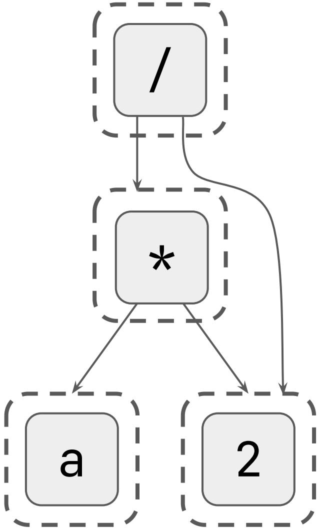
This e-class represents

(a * 2) / 2



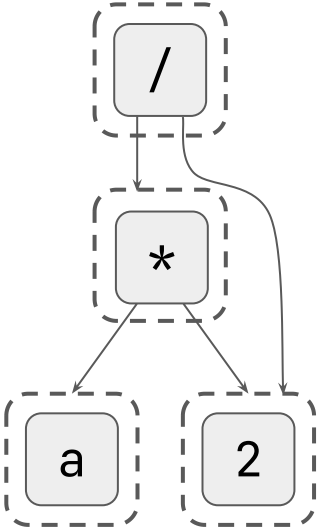
Datalog EqSat egglog eggcc

e-graphs: Applying Rules



x * 2 ⟹ x << 1

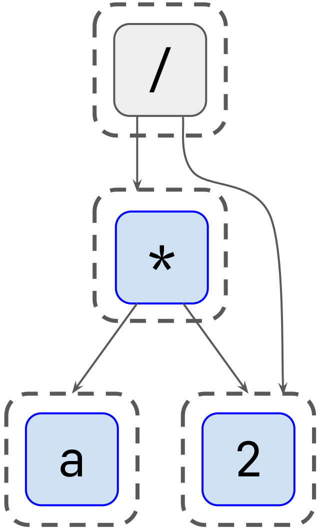
e-graphs: Applying Rules



x * 2 ⟹ x << 1

Find a term that looks like the left,
Add a term that looks like the right,
And mark them equivalent

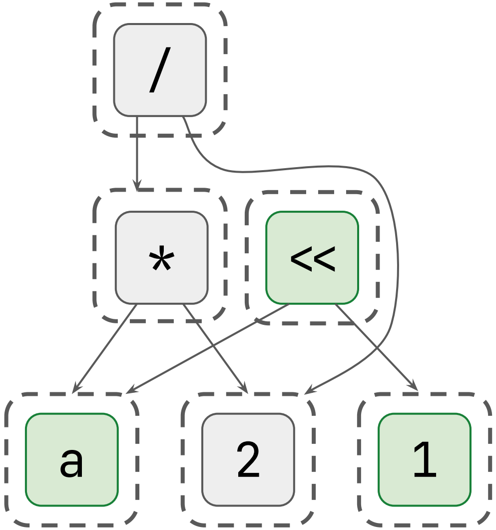
e-graphs: Applying Rules



x * 2 ⟹ x << 1

Find a term that looks like the left,
Add a term that looks like the right,
And mark them equivalent

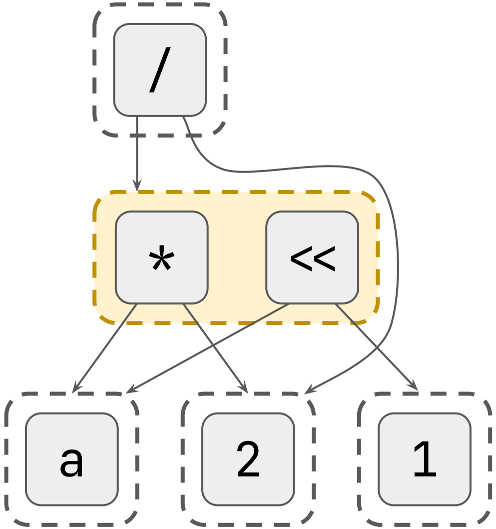
e-graphs: Applying Rules



x * 2 ⟹ x << 1

Find a term that looks like the left,
Add a term that looks like the right,
And mark them equivalent

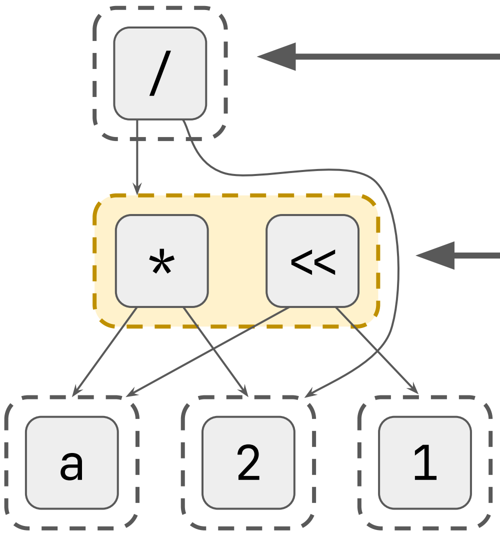
e-graphs: Applying Rules



$x * 2 \Longrightarrow x \ll 1$

Find a term that looks like the left,
Add a term that looks like the right,
And mark them equivalent

e-graphs: Applying Rules

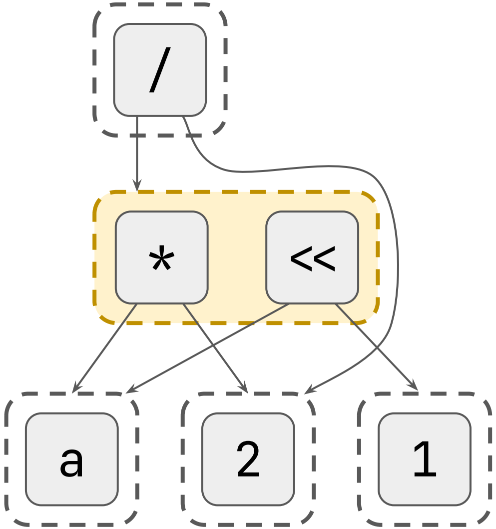


This e-class represents
`(a * 2)/2` **and** `(a << 1)/2`

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

$x * 2 \Longrightarrow x \ll 1$

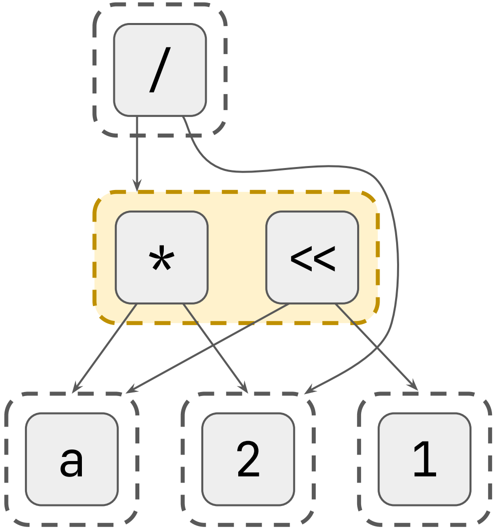
e-graphs: Applying Rules



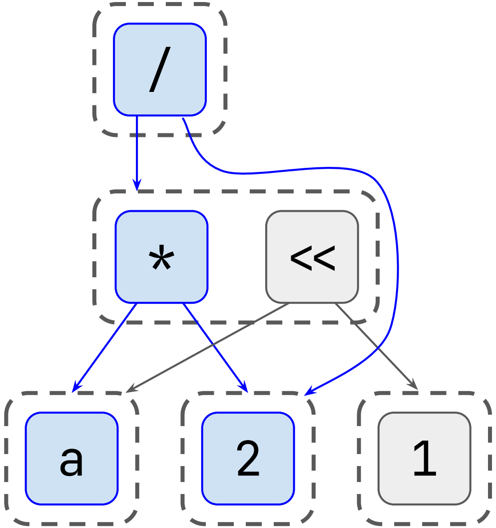
x * 2 ⟹ x << 1

(x * y)/z ⟹ x * (y/z)

e-graphs: Applying Rules

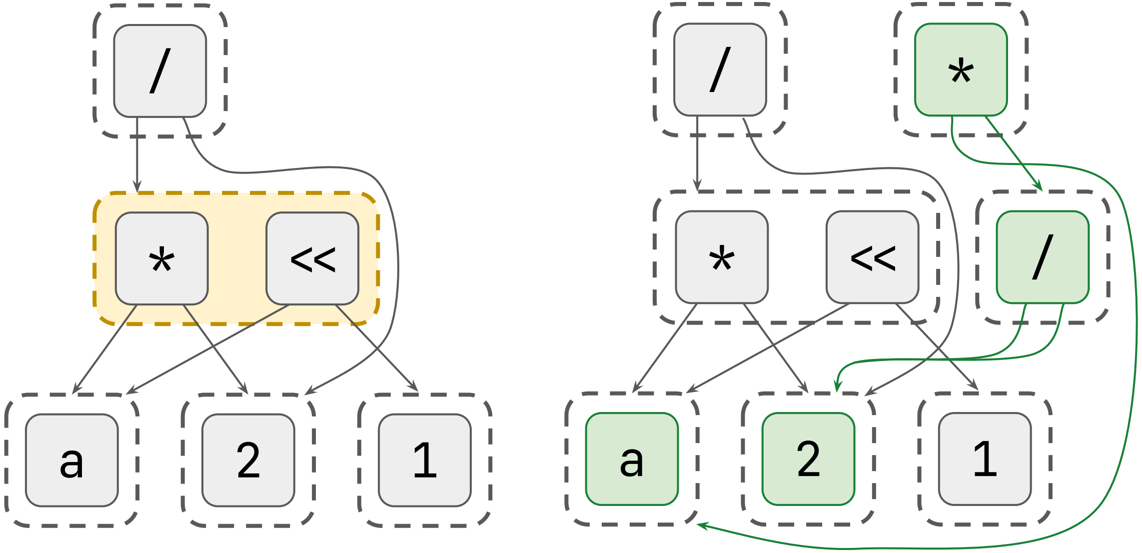


x * 2 ⟹ x << 1



(x * y)/z ⟹ x * (y/z)

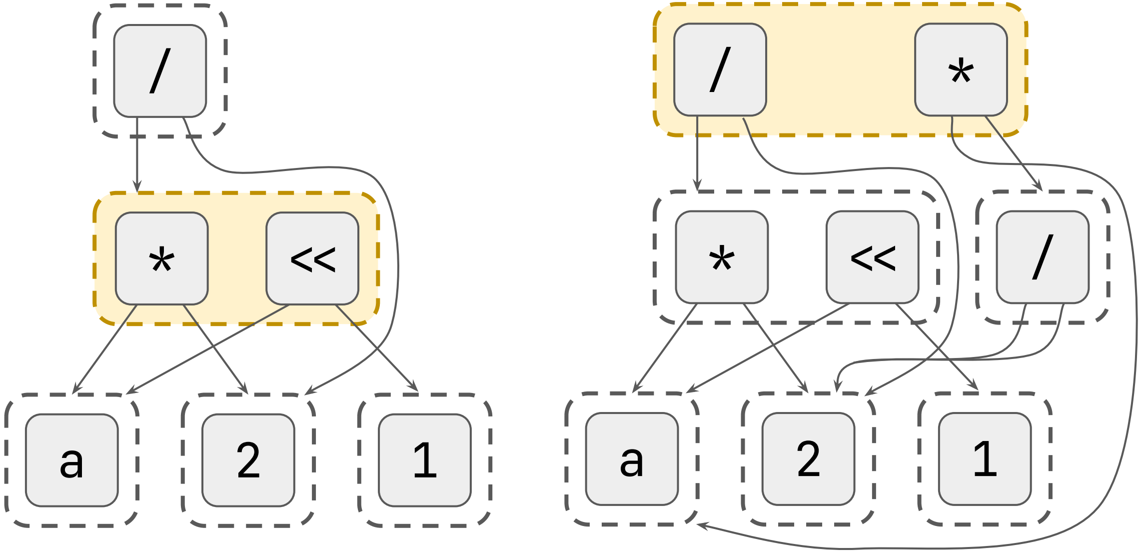
e-graphs: Applying Rules



x * 2 ⟹ x << 1

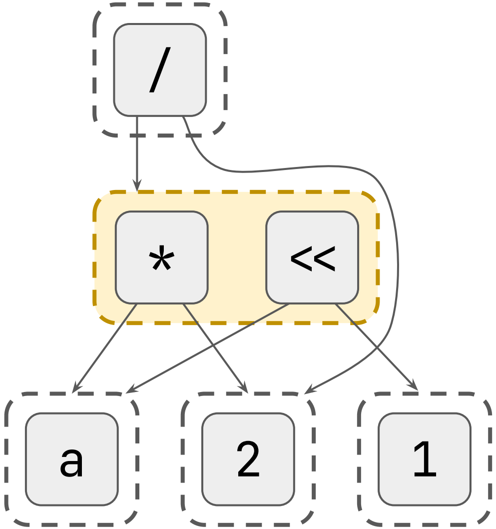
(x * y)/z ⟹ x * (y/z)

e-graphs: Applying Rules

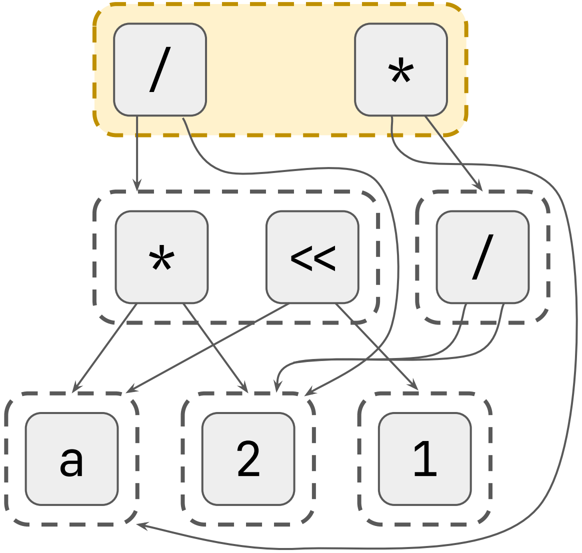


x * 2 ⟹ x << 1   (x * y)/z ⟹ x * (y/z)

e-graphs: Applying Rules



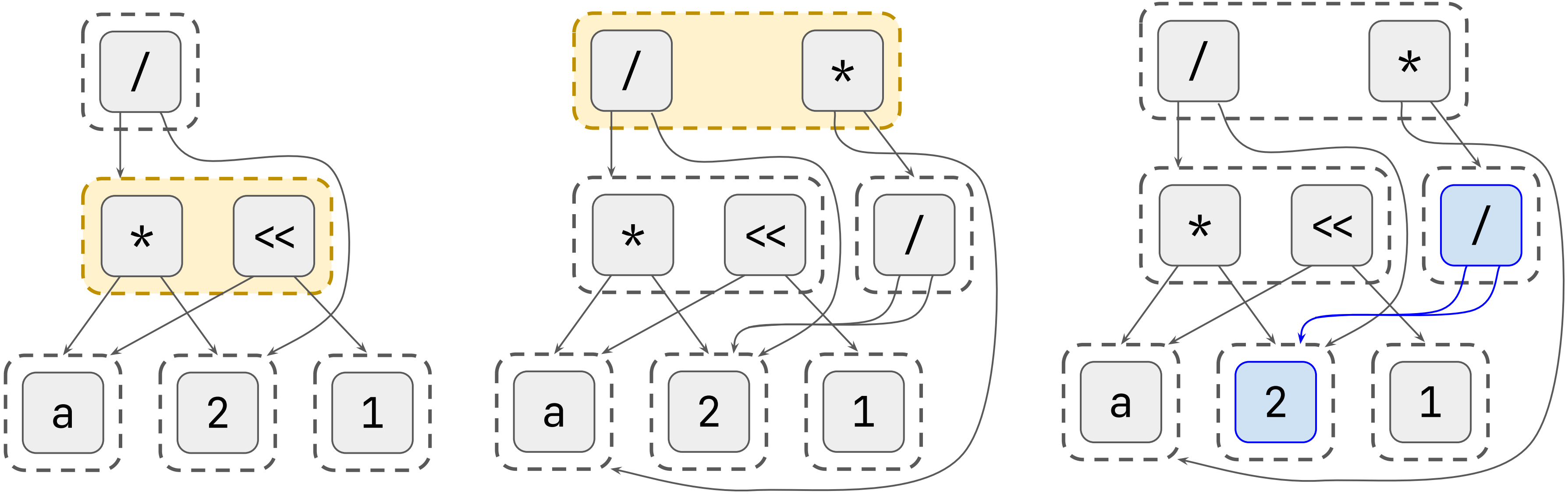
x * 2 ⟹ x << 1



(x * y)/z ⟹ x * (y/z)

x / x ⟹ 1
x * 1 ⟹ x

e-graphs: Applying Rules

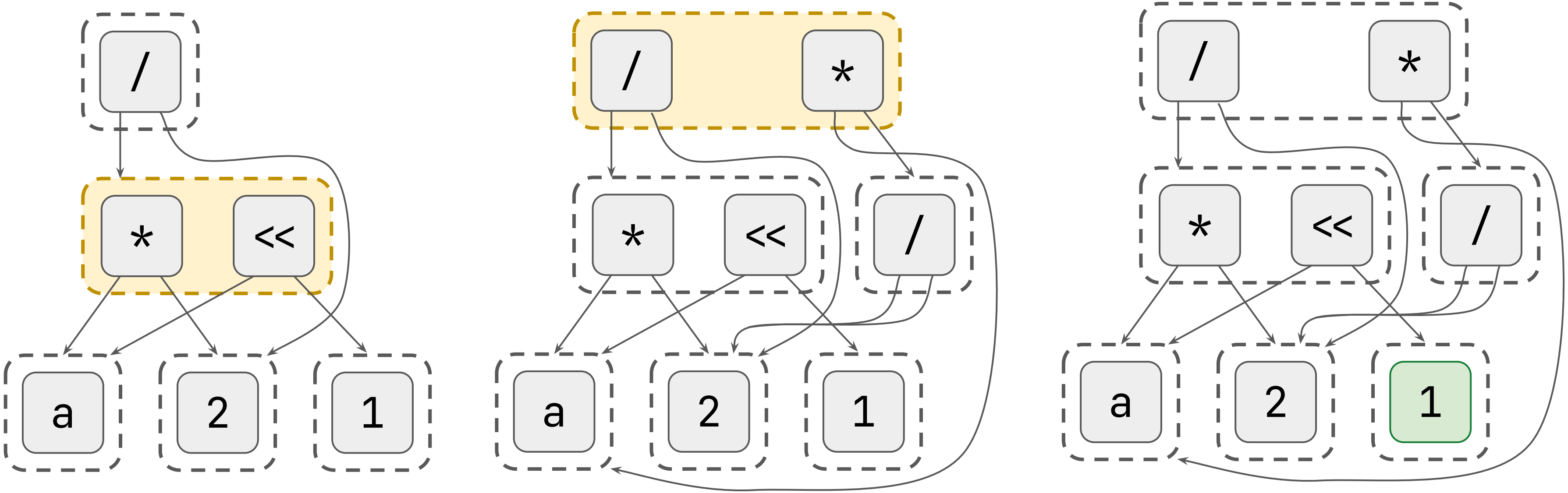


x * 2 ⟹ x << 1

(x * y)/z ⟹ x * (y/z)

x / x ⟹ 1
x * 1 ⟹ x

e-graphs: Applying Rules

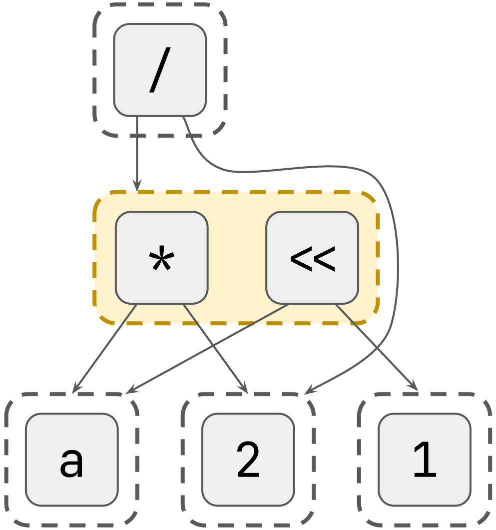


x * 2 ⟹ x << 1

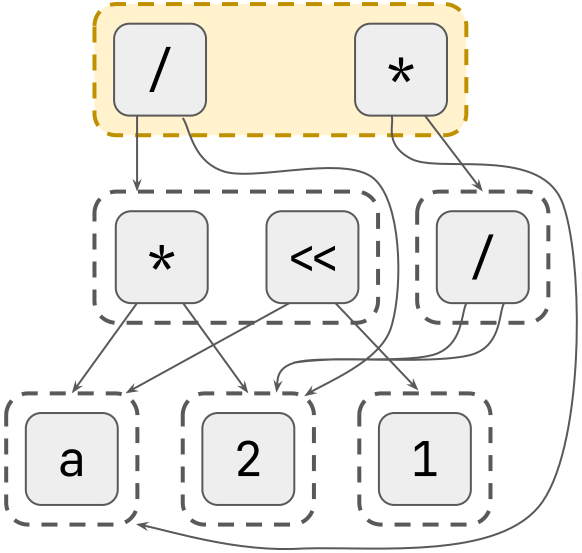
(x * y)/z ⟹ x * (y/z)

x / x ⟹ 1
x * 1 ⟹ x

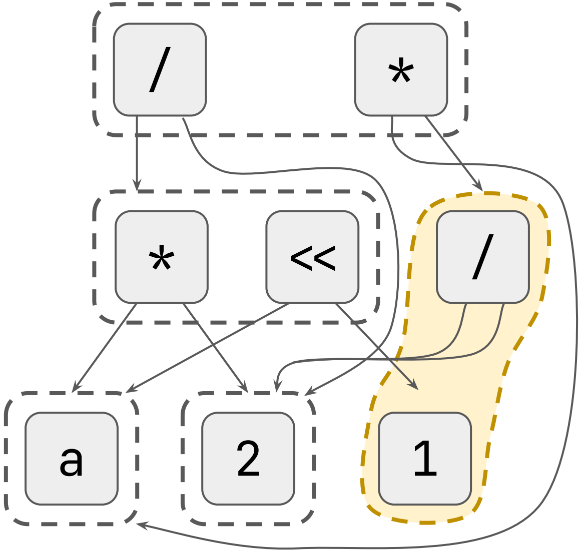
e-graphs: Applying Rules



x * 2 ⟹ x << 1

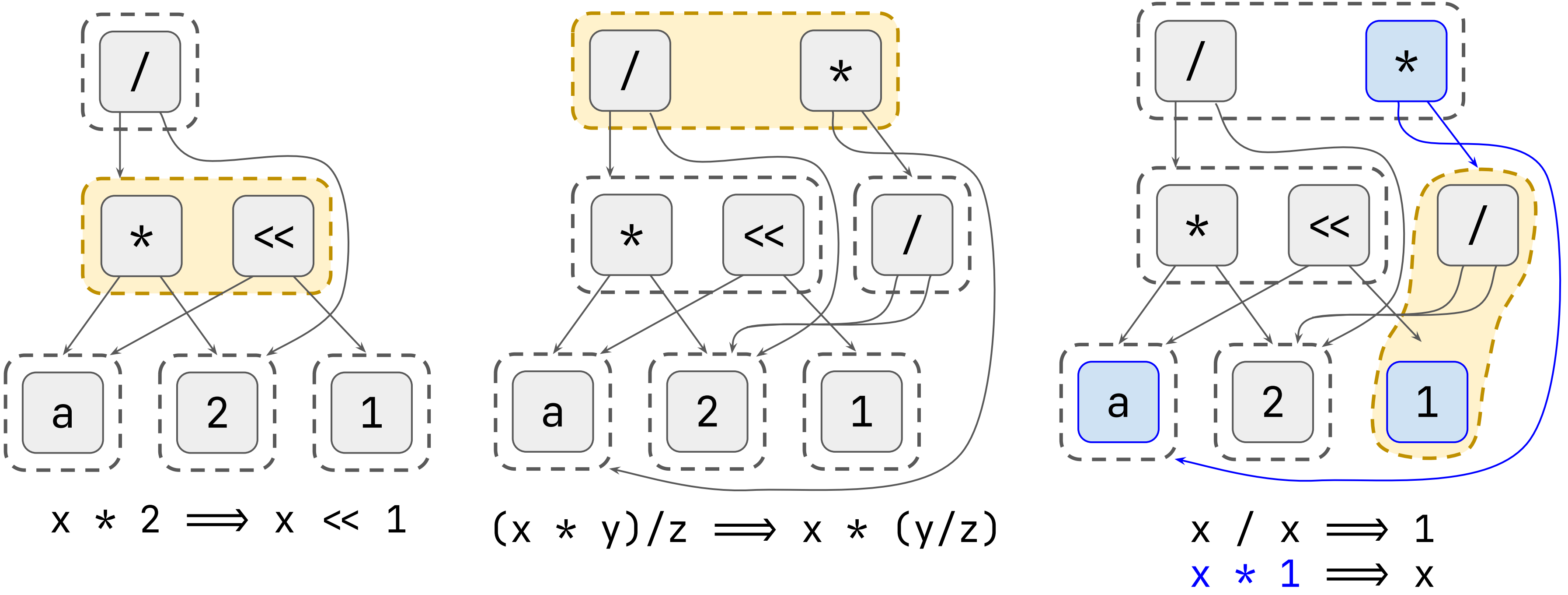


(x * y)/z ⟹ x * (y/z)

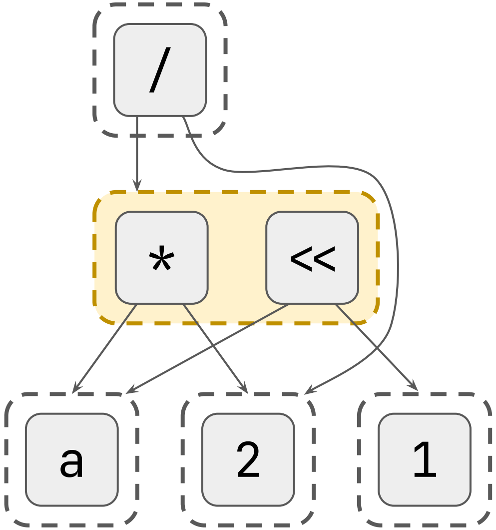


x / x ⟹ 1
x * 1 ⟹ x

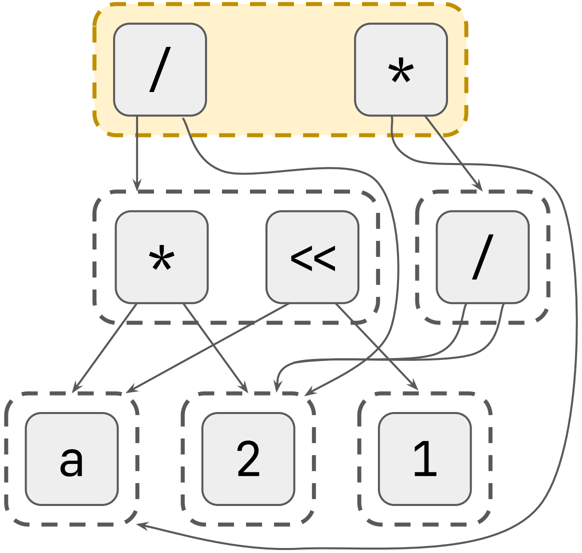
e-graphs: Applying Rules



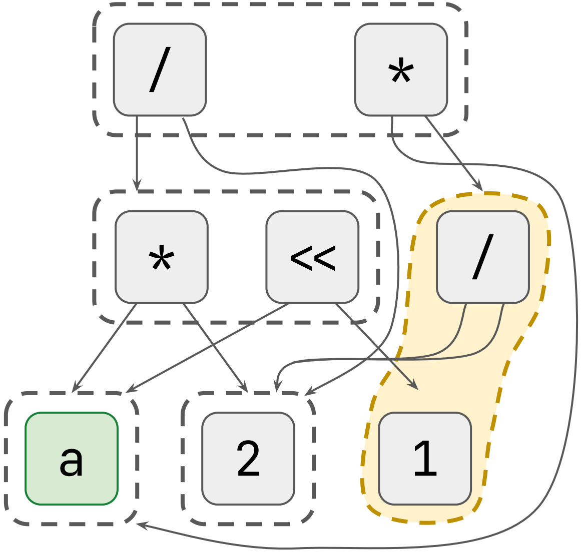
e-graphs: Applying Rules



x * 2 ⟹ x << 1



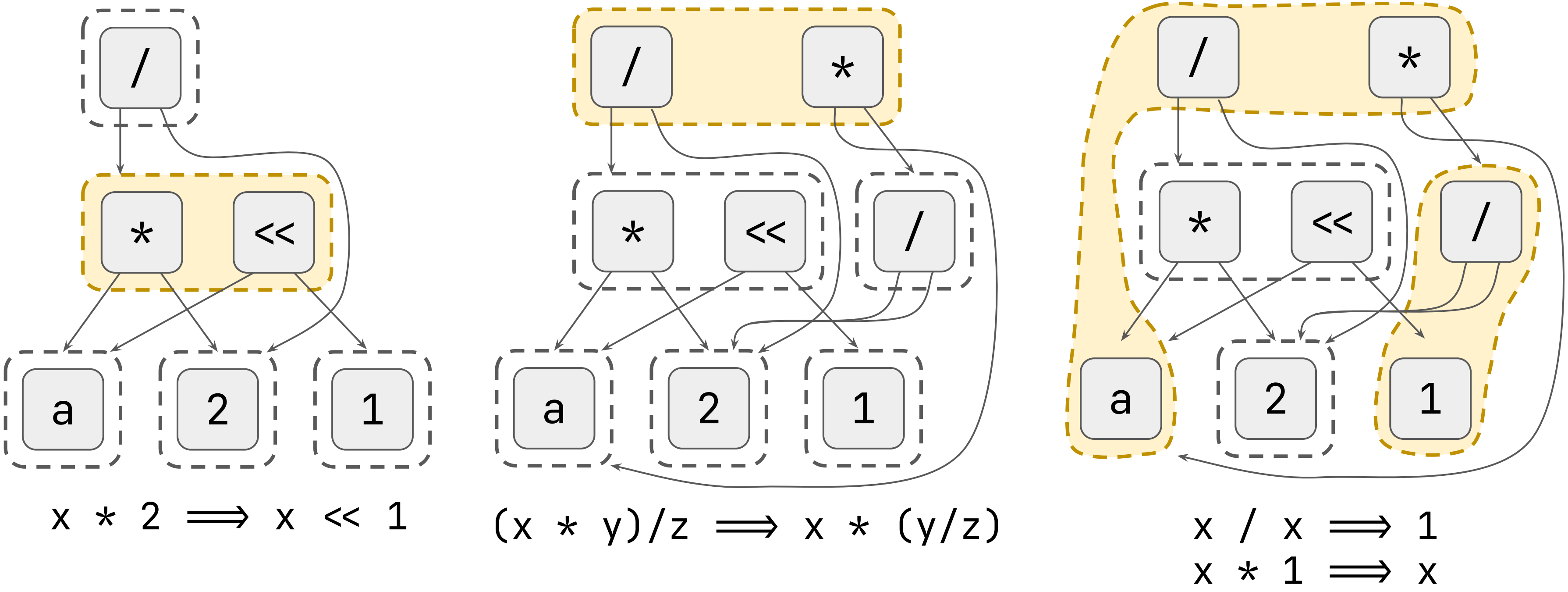
(x * y)/z ⟹ x * (y/z)



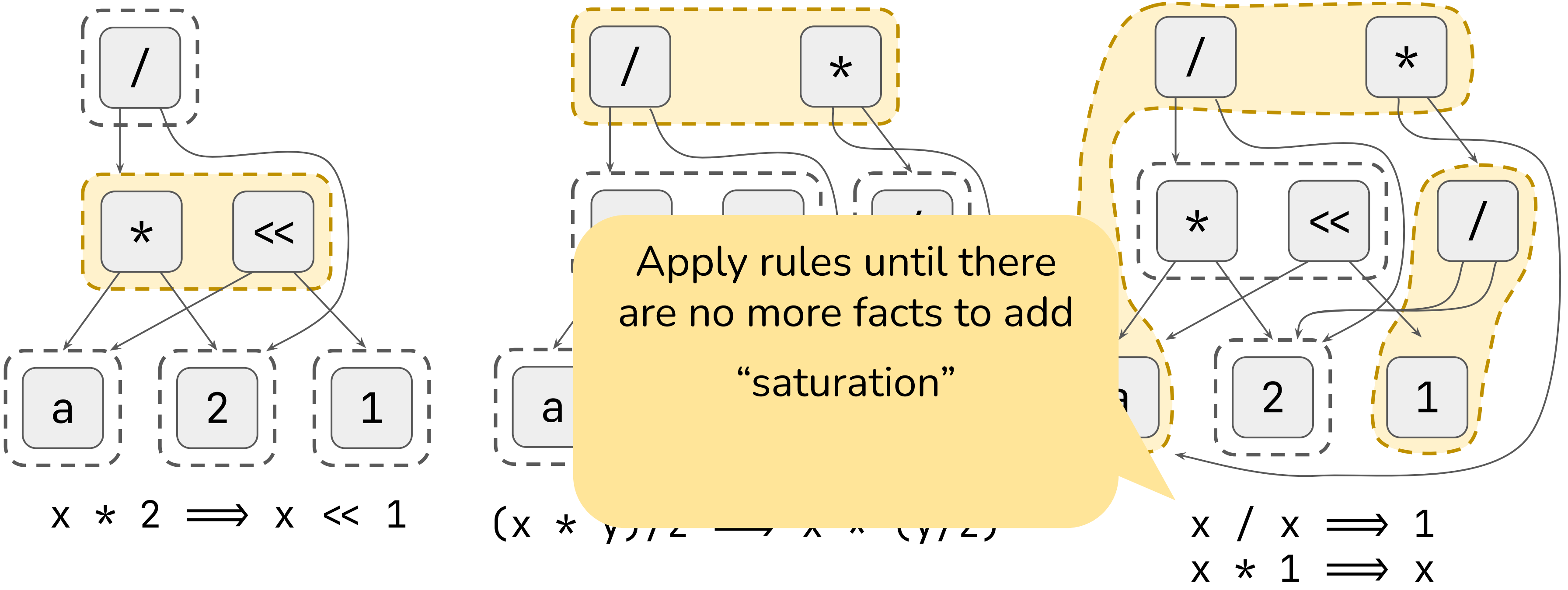
x / x ⟹ 1

x * 1 ⟹ x

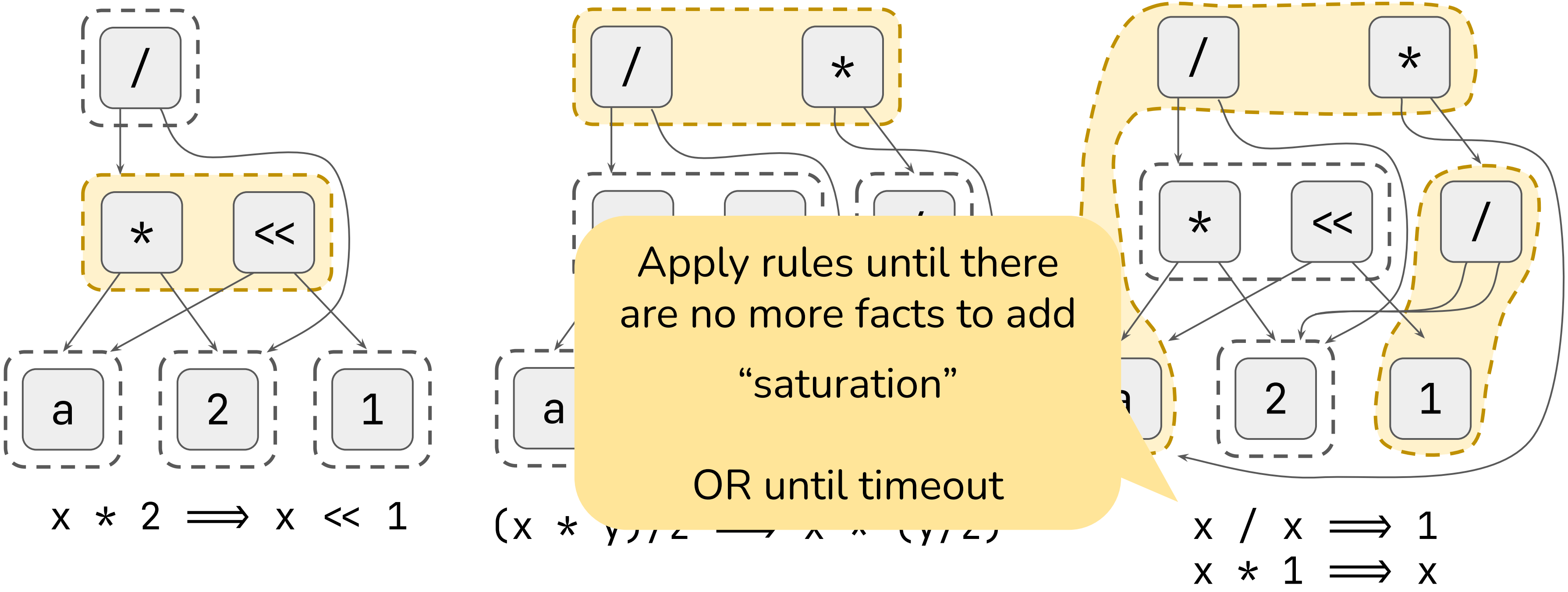
e-graphs: Applying Rules



e-graphs: Applying Rules



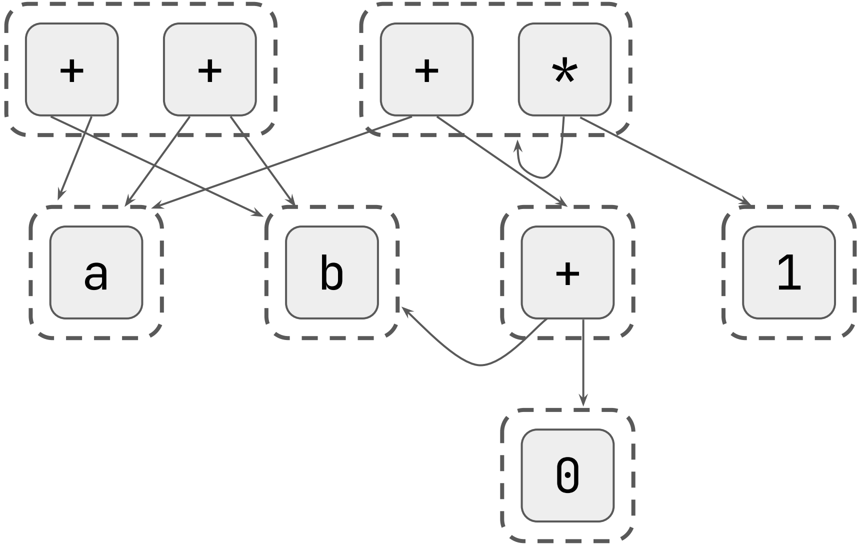
e-graphs: Applying Rules



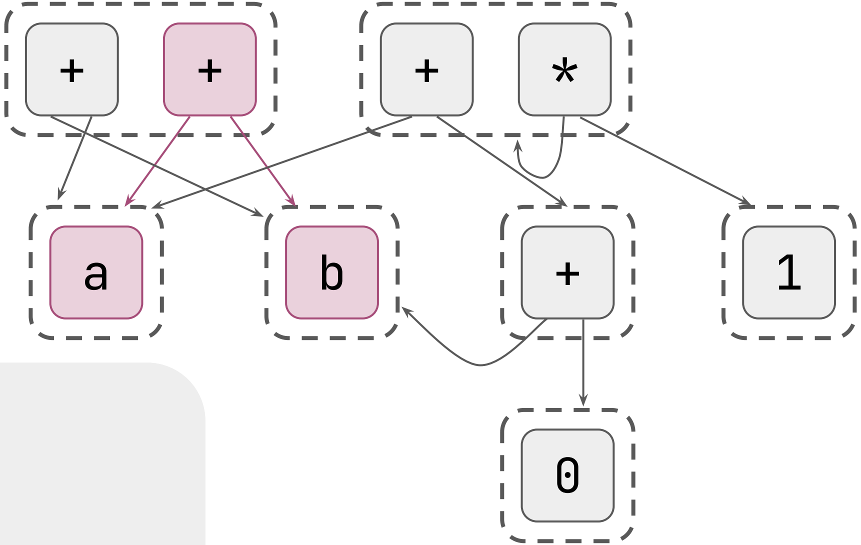
Congruence

a = b implies f(a) = f(b)

e-graphs: Congruence

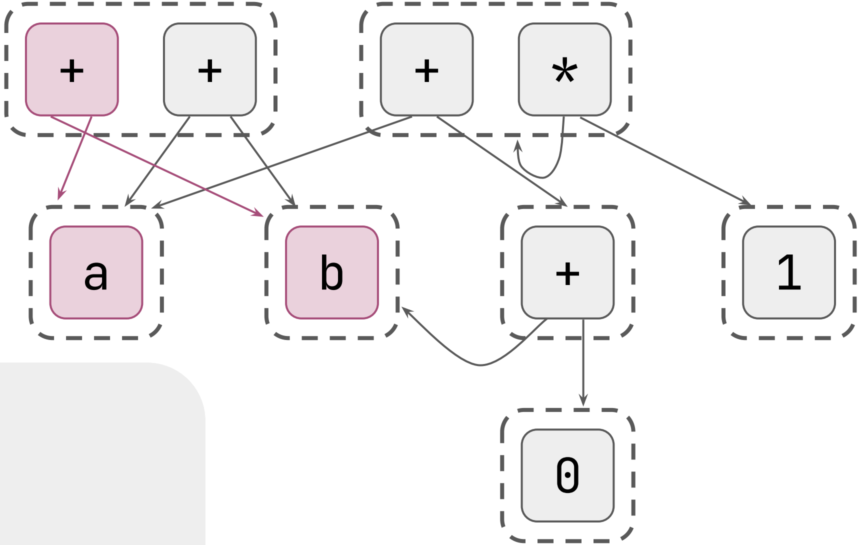


e-graphs: Congruence



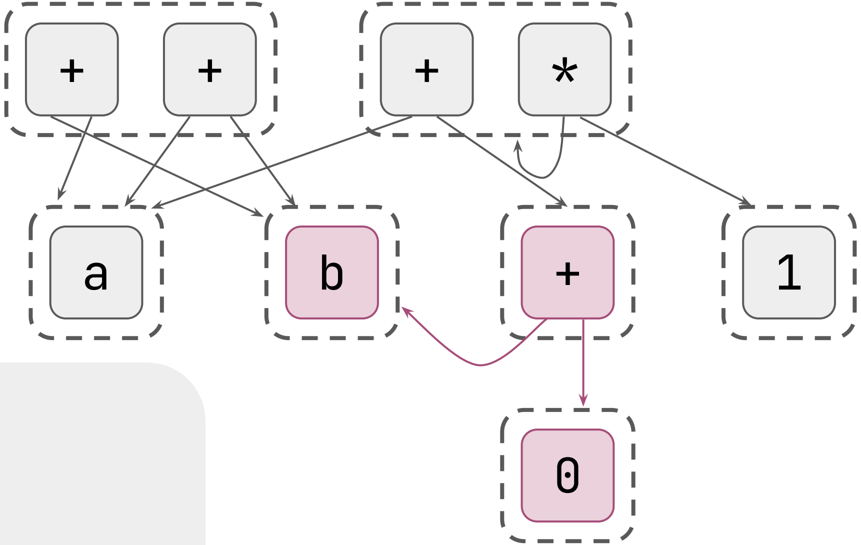

```
a + b
```

e-graphs: Congruence



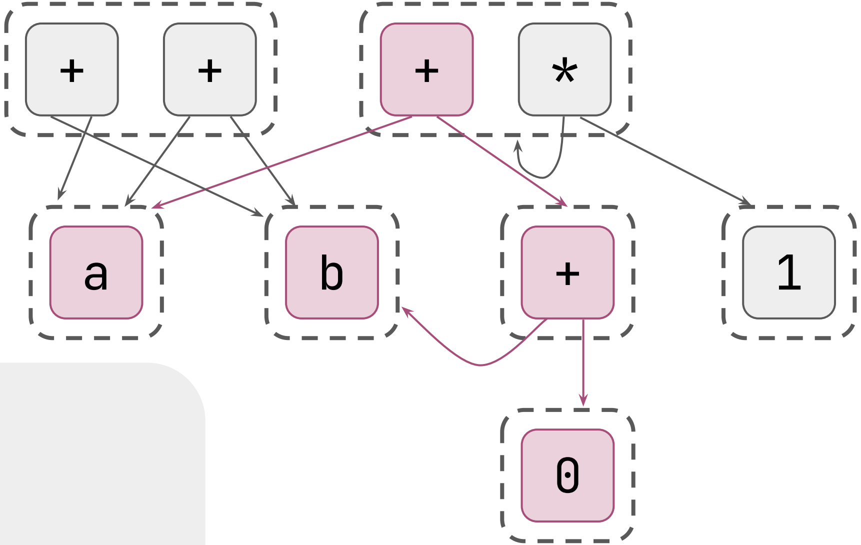

```
a + b
b + a
```

e-graphs: Congruence



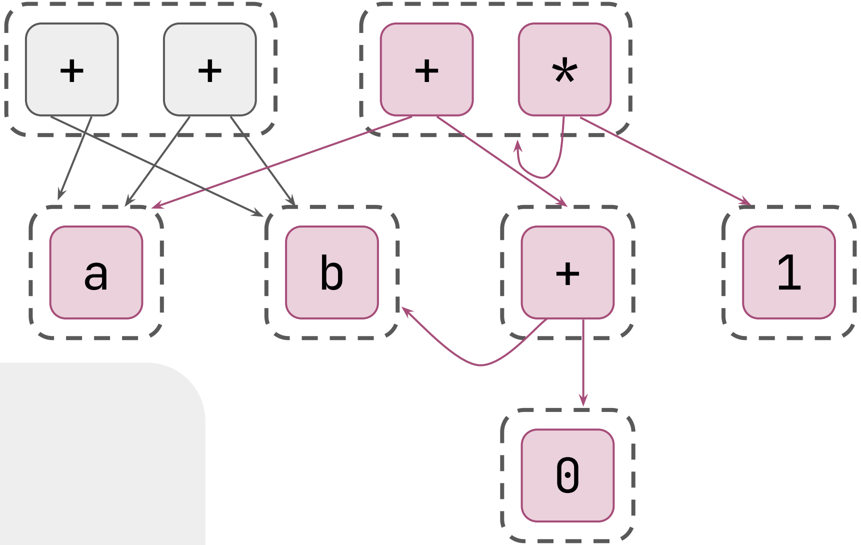

```
a + b
b + a
b + 0
```

e-graphs: Congruence



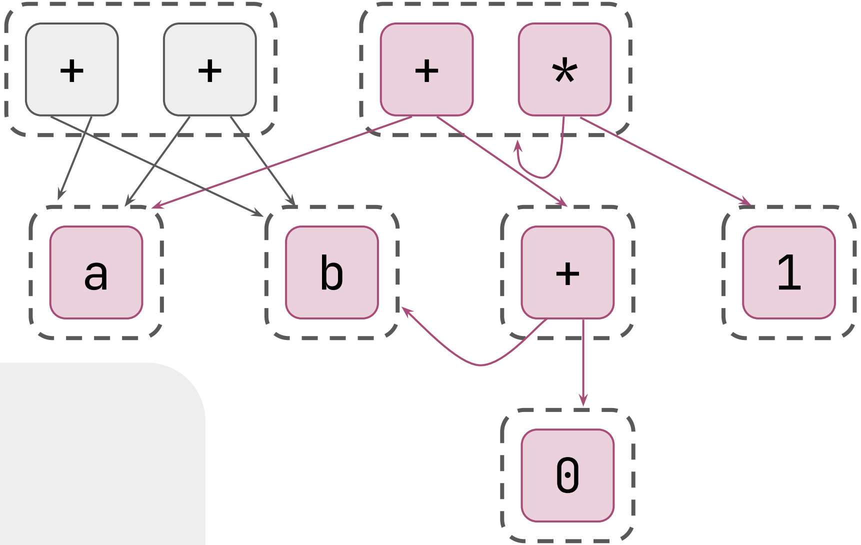

```
a + b
b + a
b + 0
a + (b + 0)
```

e-graphs: Congruence



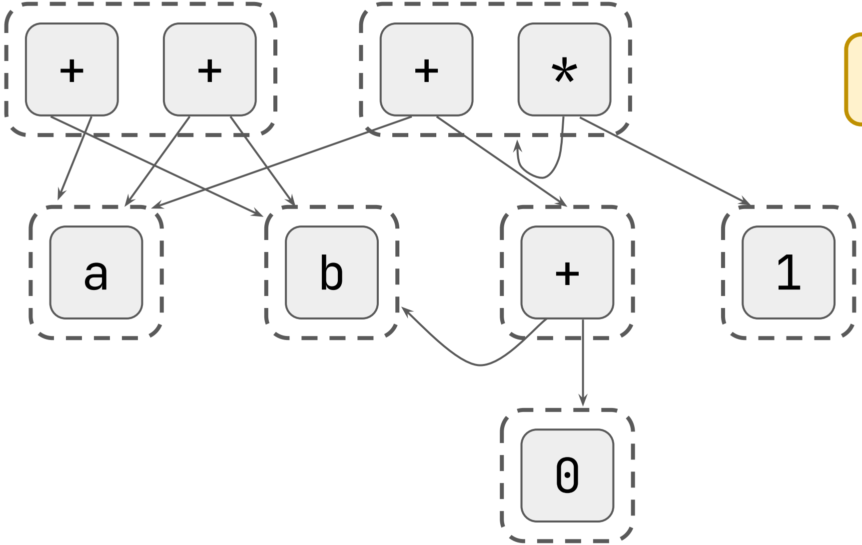

```
 a + b
 b + a
 b + 0
 a + (b + 0)
(a + (b + 0)) * 1
```

e-graphs: Congruence



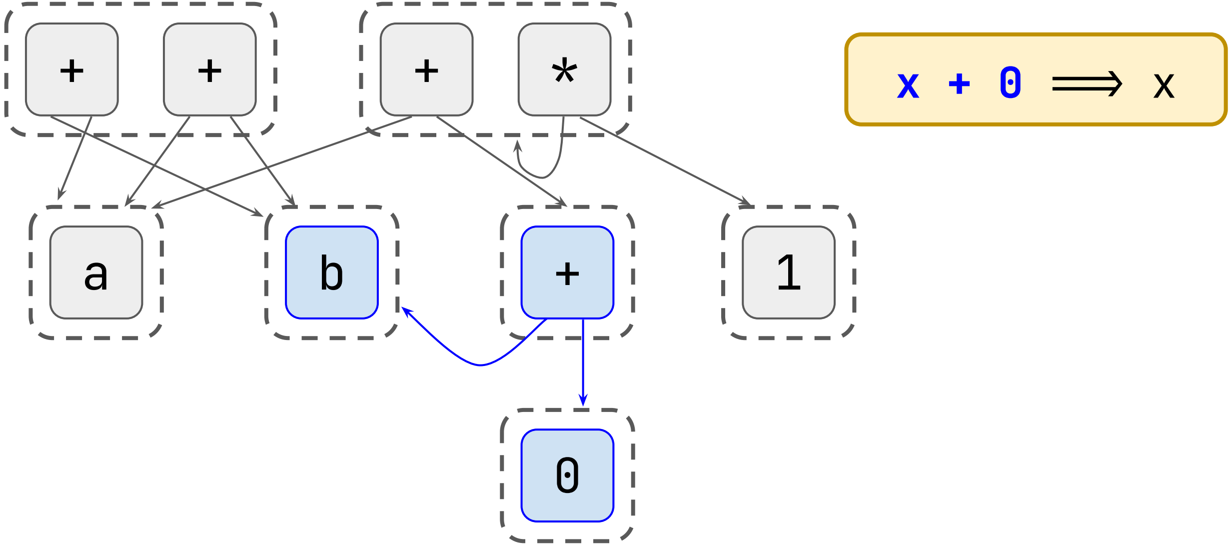

```
  a + b
  b + a
  b + 0
  a + (b + 0)
(a + (b + 0)) * 1
((a + (b + 0)) * 1) * 1
```

e-graphs: Congruence

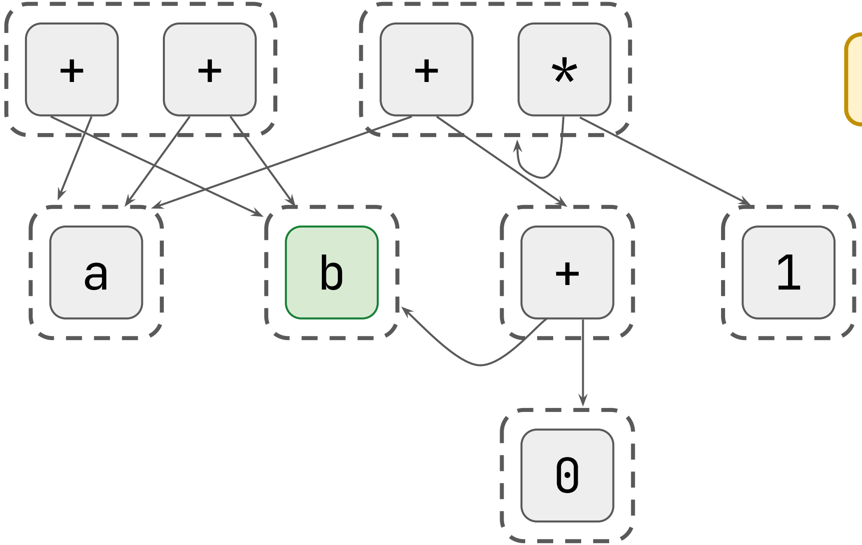


$x + 0 \Longrightarrow x$

e-graphs: Congruence

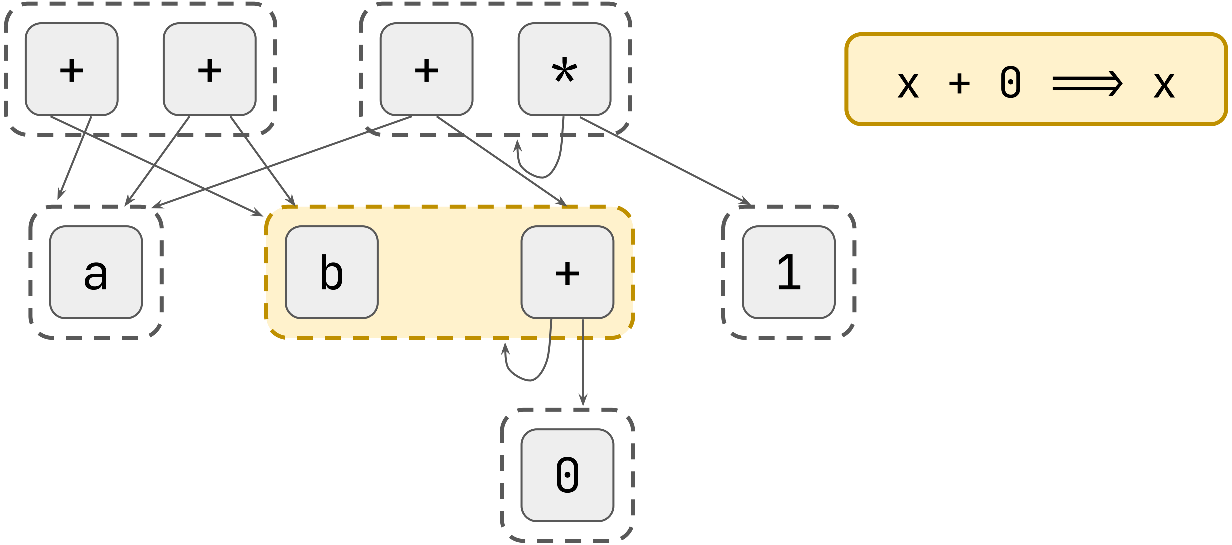


e-graphs: Congruence

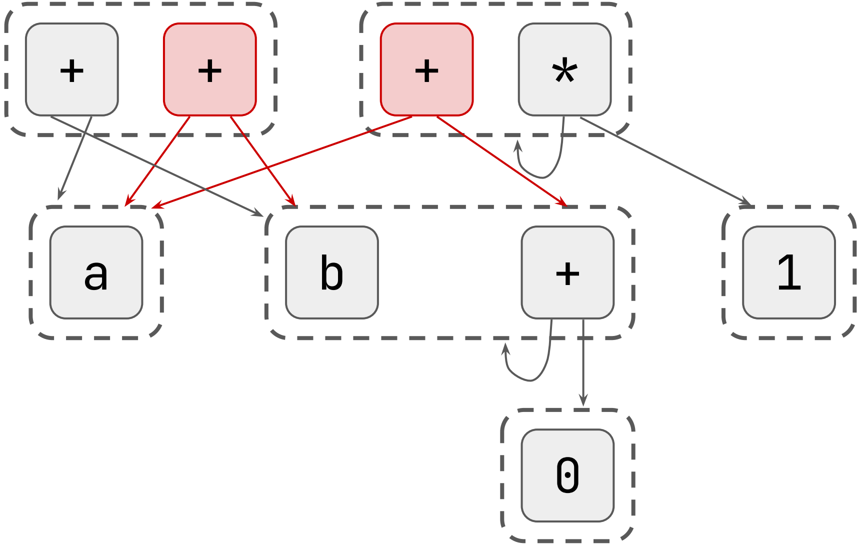


x + 0 ⟹ x

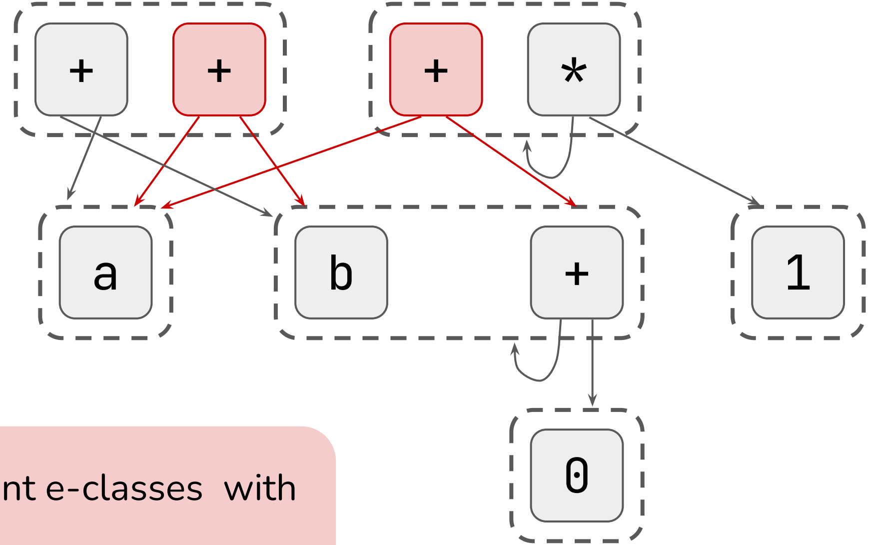
e-graphs: Congruence



e-graphs: Congruence

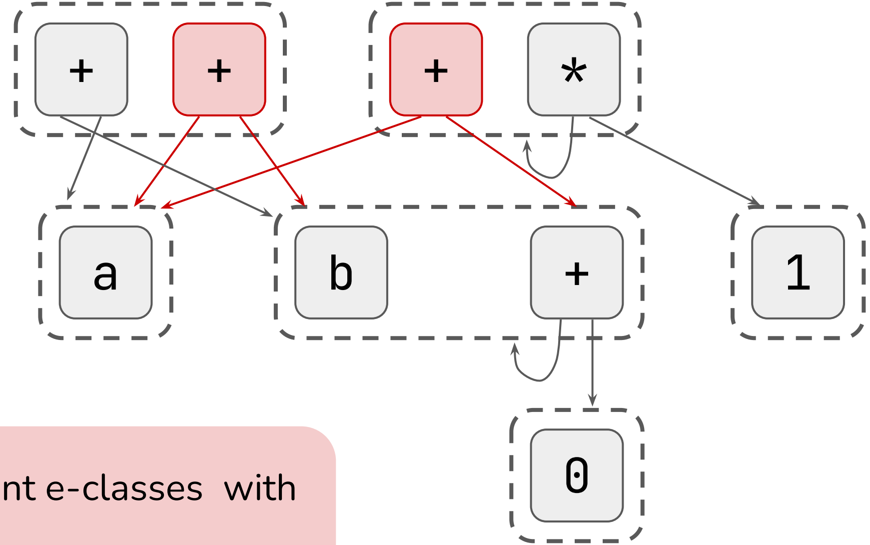


e-graphs: Congruence



Two e-nodes in different e-classes with
- same operator
- same children

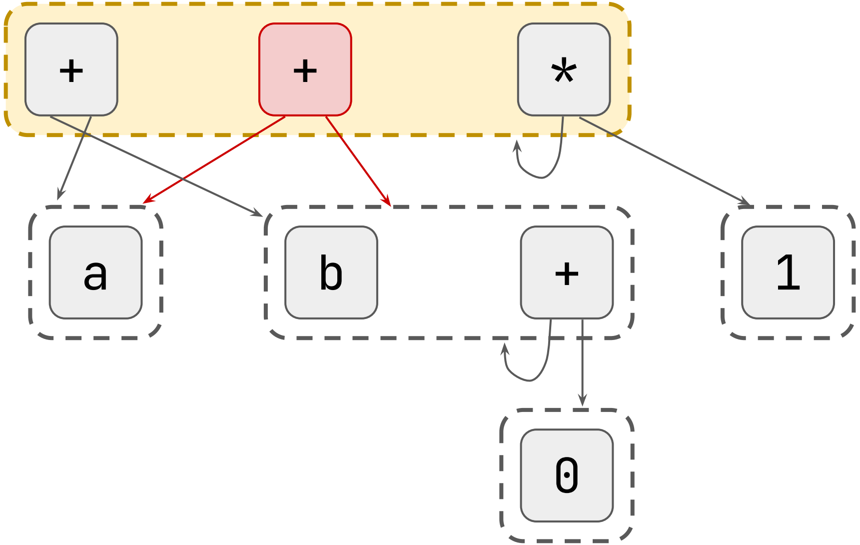
e-graphs: Congruence



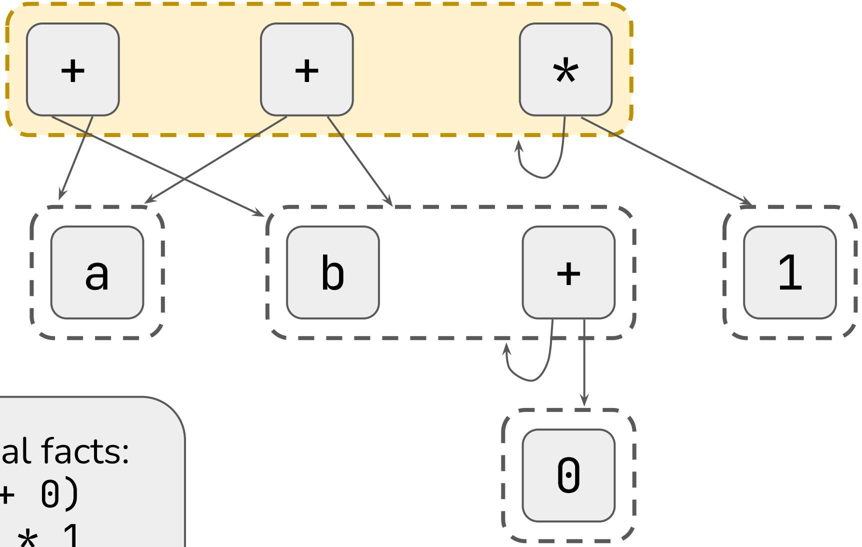
Two e-nodes in different e-classes with
- same operator
- same children

we should merge them!

e-graphs: Congruence



e-graphs: Congruence



Now we know additional facts:
- b + a = a + (b + 0)
- a + b = (b + a) * 1
- (b + 0) + a = a + b

...

e-graphs: Summary

- E-graphs compactly represent many equivalent terms
- Rewrite rules have the form `lhs` $\implies$ `rhs`
- Equality saturation grows an e-graph by applying rewrite rules
  - Until saturation or timeout
- E-graphs maintain congruence

egglog is a fixpoint reasoning system that unifies Datalog and Equality Saturation

egglog

```
(rule (query)
      (action))
```

egglog

```
(rule (query)
      (action))
```

Match on one or more terms in the database

egglog

```
(rule (query)
      (action))
```

Match on one or more terms in the database

Add new terms and/or mark terms equivalent

egglog

```
Ancestor(X, Y):- Parent(X, Y)
Ancestor(X, Z):-
    Parent(X, Y),
    Ancestor(Y, Z)
Parent(“Alice”, “Bob”)
Parent(“Bob, “Charlie”)
```

egglog

```
(relation parent
  (String String))
(relation ancestor
  (String String))
(parent “Alice” “Bob”)
(parent “Bob” “Charlie”)
(rule ((parent x y))
      ((ancestor x y)))
(rule ((parent x y)
       (ancestor y z))
      ((ancestor x z))
(run) ; run the rules
```

```
Ancestor(X, Y):- Parent(X, Y)

Ancestor(X, Z):-

    Parent(X, Y),

    Ancestor(Y, Z)

Parent(“Alice”, “Bob”)

Parent(“Bob, “Charlie”)
```

egglog

```
(relation parent
  (String String))
(relation ancestor
  (String String))
(parent “Alice” “Bob”)
(parent “Bob” “Charlie”)
(rule ((parent x y))
      ((ancestor x y)))
(rule ((parent x y)
       (ancestor y z))
      ((ancestor x z))
(run) ; run the rules
```

Parent	Id

Ancestor	Id

egglog

```
(relation parent
  (String String))
(relation ancestor
  (String String))
(parent "Alice" "Bob")
(parent "Bob" "Charlie")
(rule ((parent x y))
      ((ancestor x y)))
(rule ((parent x y)
       (ancestor y z))
      ((ancestor x z)))
(run) ; run the rules
```

Parent		Id
Alice	Bob	1
Bob	Charlie	2

Ancestor		Id

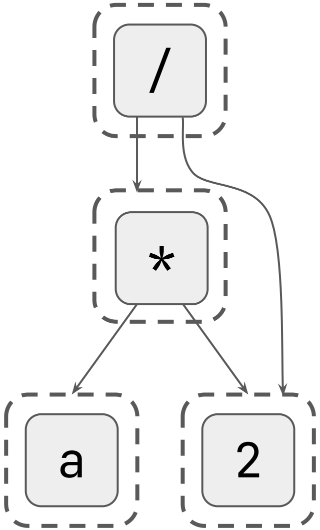
egglog

```
(relation parent
  (String String))
(relation ancestor
  (String String))
(parent “Alice” “Bob”)
(parent “Bob” “Charlie”)
(rule ((parent x y))
      ((ancestor x y)))
(rule ((parent x y)
       (ancestor y z))
      ((ancestor x z)))
(run) ; run the rules
```

Parent		Id
Alice	Bob	1
Bob	Charlie	2

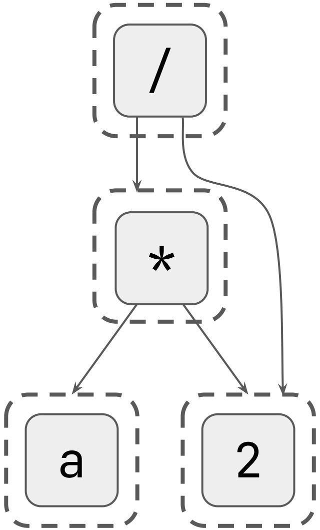
Ancestor		Id
Alice	Bob	3
Bob	Charlie	4
Alice	Charlie	5

egglog



egglog

```
; Declare a datatype
(datatype Expr
  (Var String)
  (Num i64)
  (Mul Expr Expr)
  (Div Expr Expr))

; Add a term
(Div (Mul (Var “a”) (Num 2))
     (Num 2))
```



egglog

```
; Declare a datatype
(datatype Expr
  (Var String)
  (Num i64)
  (Mul Expr Expr)
  (Div Expr Expr))

; Add a term
(Div (Mul (Var “a”) (Num 2))
     (Num 2))
```

Div		Id
id 3	id 1	4

Mul		Id
id 2	id 1	3

Var	Id
“a”	2

Num	Id
2	1

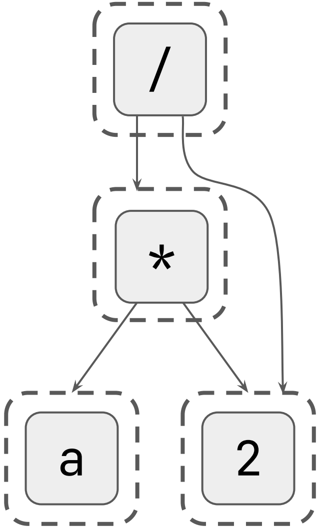
egglog

```
; Declare a datatype
(datatype Expr
  (Var String)
  (Num i64)
  (Mul Expr Expr)
  (Div Expr Expr))

; Add a term
(Div (Mul (Var “a”) (Num 2))
     (Num 2))
```

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

$$x / x \Longrightarrow 1$$

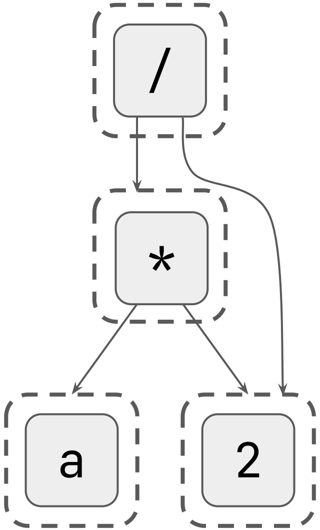
$$x * 1 \Longrightarrow x$$



egglog

```
; Declare rules
(rule
  ((= e (Div (Mul x y) z)))
  ((union e (Mul x (Div y z)))))

(rule
  ((= e (Div x x)))
  ((union e (Num 1))))

(rule
  ((= e (Mul x (Num 1))))
  ((union e x)))
(run) ; run the rules
```

$$(x * y) / z \Longrightarrow x * (y / z)$$
$$x / x \Longrightarrow 1$$
$$x * 1 \Longrightarrow x$$



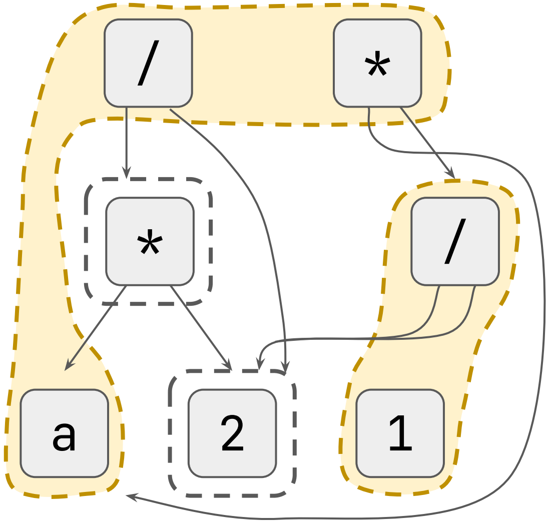
egglog

```
; Declare rules
(rule
  ((= e (Div (Mul x y) z)))
  ((union e (Mul x (Div y z)))))

(rule
  ((= e (Div x x)))
  ((union e (Num 1))))

(rule
  ((= e (Mul x (Num 1))))
  ((union e x)))
(run) ; run the rules
```

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

$$x / x \Longrightarrow 1$$

$$x * 1 \Longrightarrow x$$



egglog subsumes Datalog and Equality Saturation

Rules can match on multiple facts, like in Datalog

egglog subsumes Datalog and Equality Saturation

Rules can match on multiple facts, like in Datalog

egglog subsumes Datalog and Equality Saturation

Rules can mark terms equivalent, like in Equality Saturation

Now let's build a compiler!

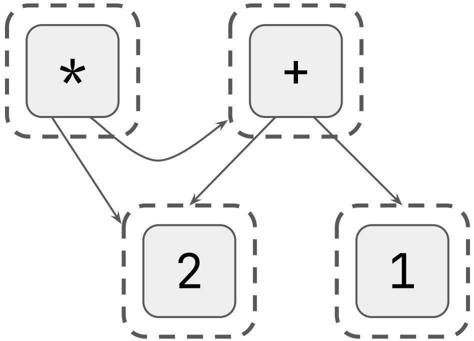
Optimization: Constant Folding

```
x = (1 + 2 + 3) * (5 + 6 + 7)
print x
```

```
print 108
```

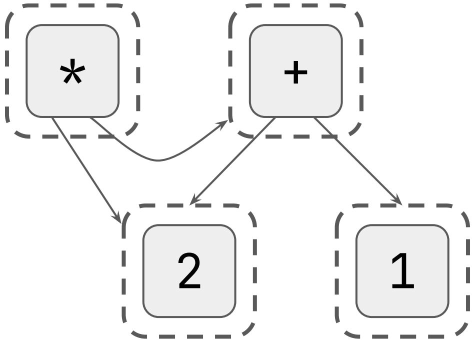
Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))
```



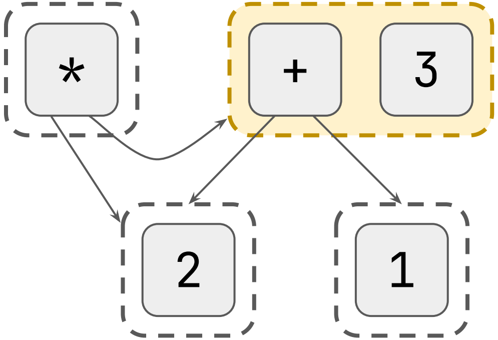
Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))
```



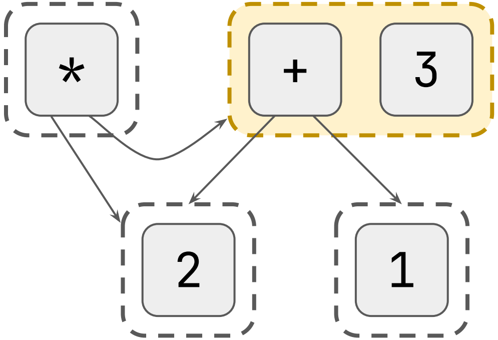
Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))
```



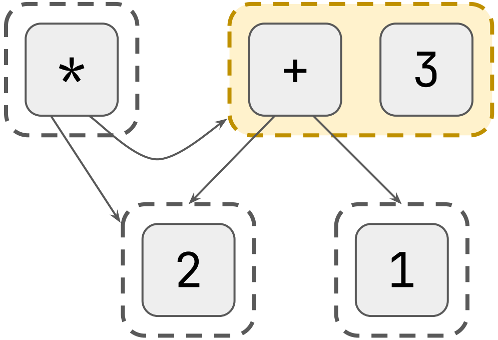
Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))

(rule ((= e (Mul (Num x) (Num y))))
      ((union e (Num (* x y)))))
```



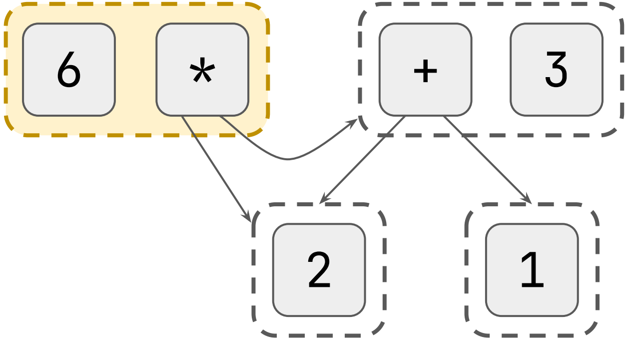
Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))

(rule ((= e (Mul (Num x) (Num y))))
      ((union e (Num (* x y)))))
```



Optimization: Constant Folding

```
(rule ((= e (Add (Num x) (Num y))))
      ((union e (Num (+ x y)))))

(rule ((= e (Mul (Num x) (Num y))))
      ((union e (Num (* x y)))))
```



Optimization: Interval Analysis

```
x = ... // between 2-4
y = ... // between 3-5
print (x + y < 100)
```

```
print true
```

Optimization: Interval Analysis

```
(datatype Expr ...)

(datatype Interval
  (IntI i64 i64)
  (BoolI bool bool))
(function ival (Expr) Interval)
```

Optimization: Interval Analysis

```
(datatype Expr ...)

(datatype Interval
  (IntI i64 i64)
  (BoolI bool bool))
(function ival (Expr) Interval)
```

```
(let one (Num 1))
(set (ival one) (IntI 1 1))
```

ival	
(Num 1)	(IntI 1 1)

Optimization: Interval Analysis

```
(datatype Expr ...)

(datatype Interval
  (IntI i64 i64)
  (BoolI bool bool))
(function ival (Expr) Interval)
```

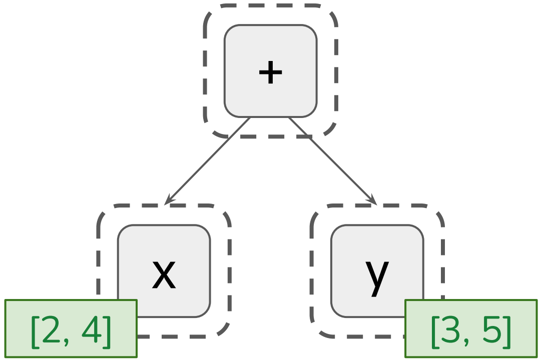
```
(let one (Num 1))
(set (ival one) (IntI 1 1))

(let t (Bool true))
(set (ival t) (BoolI true true))
```

ival	
(Num 1)	(IntI 1 1)
(Bool true)	(BoolI true true)

Optimization: Interval Analysis

```
(rule
  ((= e (Add x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (IntI (+ lo-x lo-y) (+ hi-x hi-y)))))
```

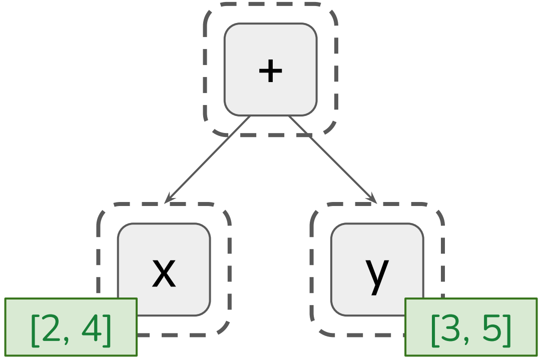
```
x = ... // between 2-4
y = ... // between 3-5
print (x + y < 100)
```



Optimization: Interval Analysis

```
(rule
  ((= e (Add x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (IntI (+ lo-x lo-y) (+ hi-x hi-y)))))
```

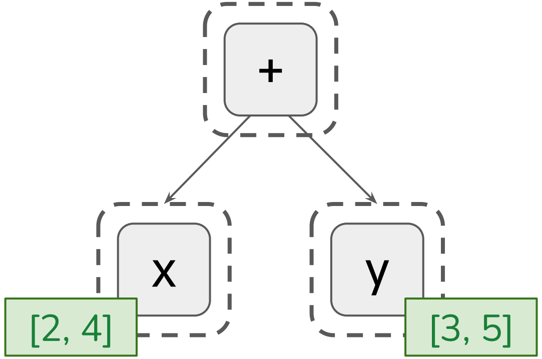
```
x = ... // between 2-4
y = ... // between 3-5
print (x + y < 100)
```



Optimization: Interval Analysis

```
(rule
  ((= e (Add x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (IntI (+ lo-x lo-y) (+ hi-x hi-y)))))
```

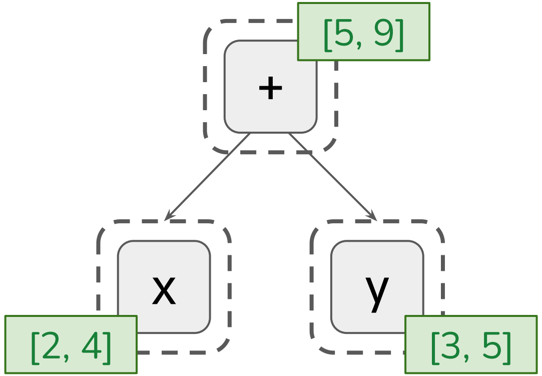
```
x = ... // between 2-4
y = ... // between 3-5
print (x + y < 100)
```



Optimization: Interval Analysis

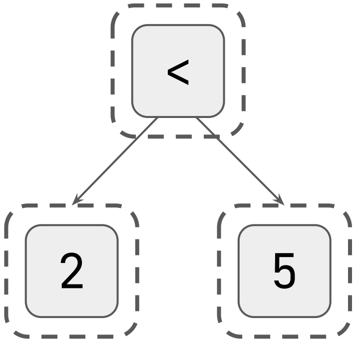
```
(rule
  ((= e (Add x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (IntI (+ lo-x lo-y) (+ hi-x hi-y)))))
```

```
x = ... // between 2-4
y = ... // between 3-5
print (x + y < 100)
```



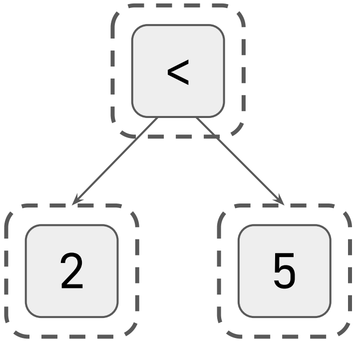
Optimization: Interval Analysis

```
(rule
  ((= e (Num x)))
  ((set (ival e) (IntI x x))))
```



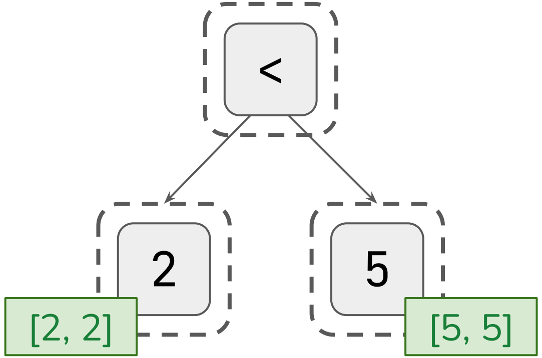
Optimization: Interval Analysis

```
(rule
  ((= e (Num x)))
  ((set (ival e) (IntI x x))))
```



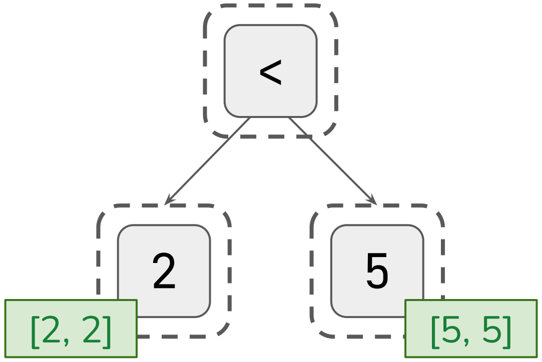
Optimization: Interval Analysis

```
(rule
  ((= e (Num x)))
  ((set (ival e) (IntI x x))))
```



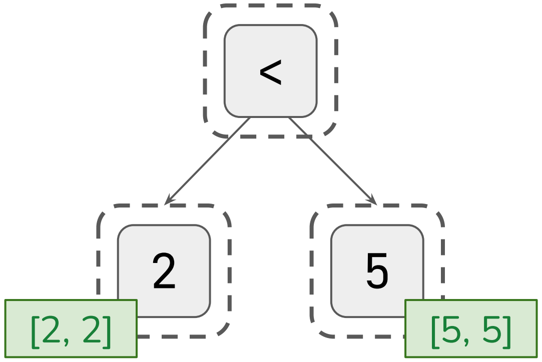
Optimization: Interval Analysis

```
(rule
  ((= e (LessThan x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```



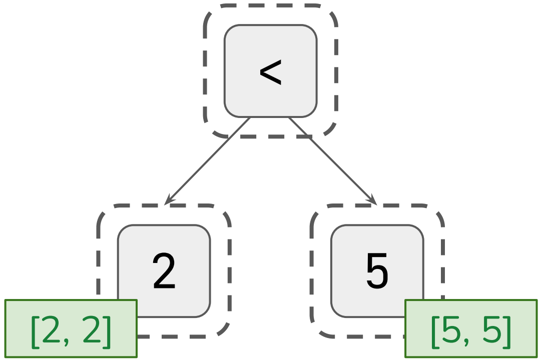
Optimization: Interval Analysis

```
(rule
  ((= e (LessThan x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```



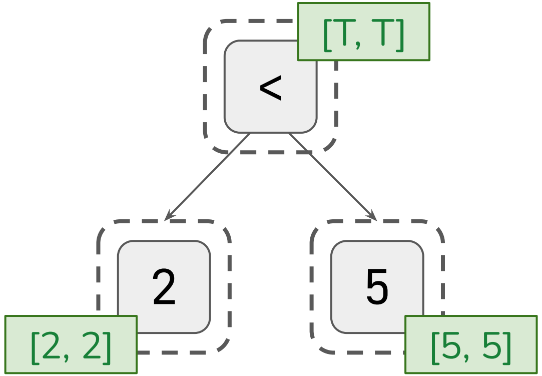
Optimization: Interval Analysis

```
(rule
  ((= e (LessThan x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```



Optimization: Interval Analysis

```
(rule
  ((= e (LessThan x y))
   (= (IntI lo-x hi-x) (ival x))
   (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```

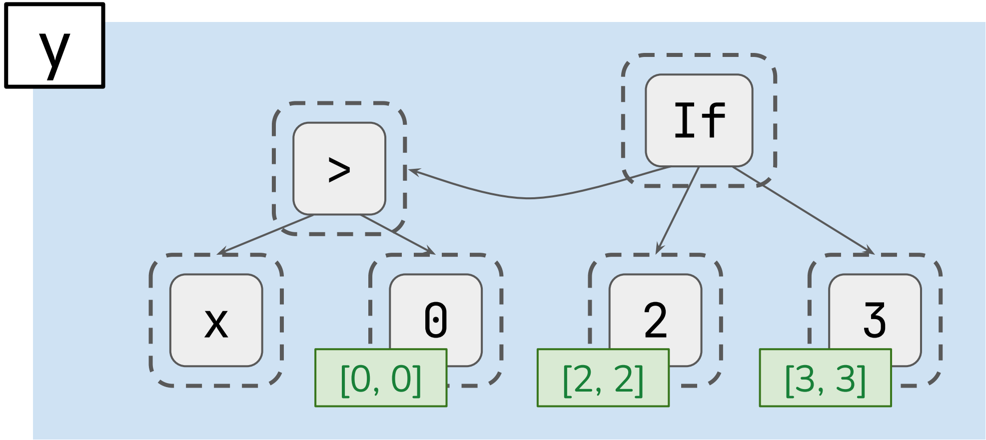


Optimization: Interval Analysis

```
if x > 0:
  y = 2
else:
  y = 3
if y < 10:
  z = 5
else:
  z = -5
return z
```

Optimization: Interval Analysis

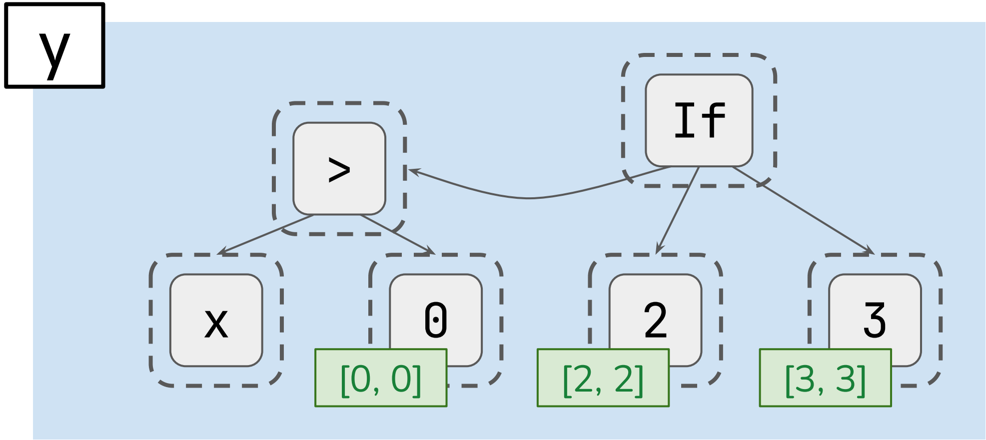
```
if x > 0:
   y = 2
else:
   y = 3
```



Optimization: Interval Analysis

```
(rule ((= e (If pred then else))
       (= then-ival (ival then))
       (= else-ival (ival else)))
  ((set (ival e) (interval-union then-ival else-ival))))
```

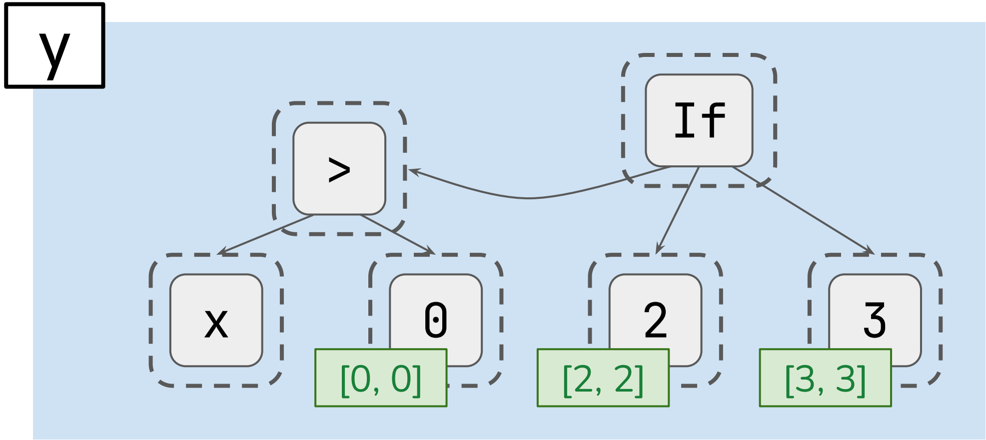
```
if x > 0:
  y = 2
else:
  y = 3
```



Optimization: Interval Analysis

```
(rule ((= e (If pred then else))
       (= then-ival (ival then))
       (= else-ival (ival else)))
  ((set (ival e) (interval-union then-ival else-ival))))
```

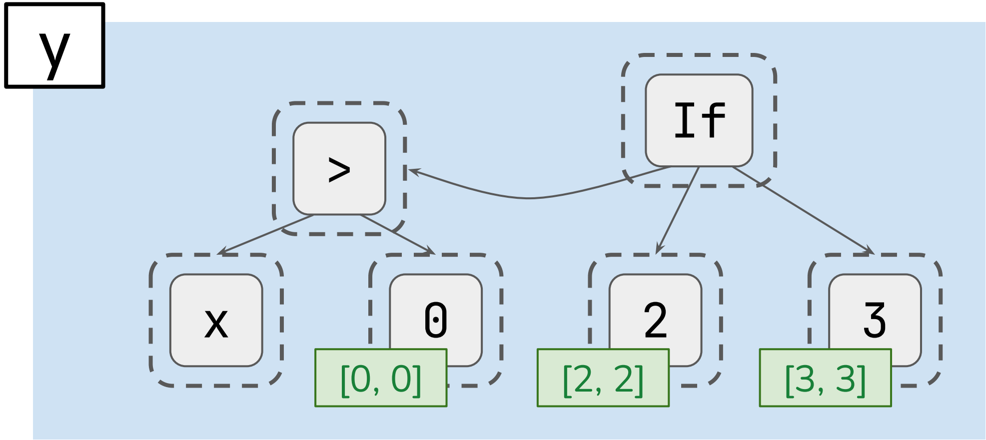
```
if x > 0:
   y = 2
else:
   y = 3
```



Optimization: Interval Analysis

```
(rule ((= e (If pred then else))
       (= then-ival (ival then))
       (= else-ival (ival else)))
  ((set (ival e) (interval-union then-ival else-ival))))
```

```
if x > 0:
  y = 2
else:
  y = 3
```

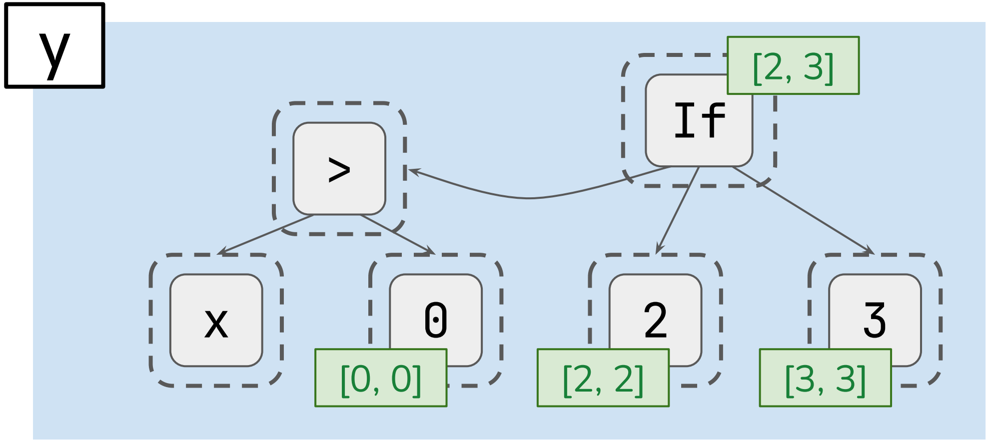


Datalog EqSat egglog eggcc

Optimization: Interval Analysis

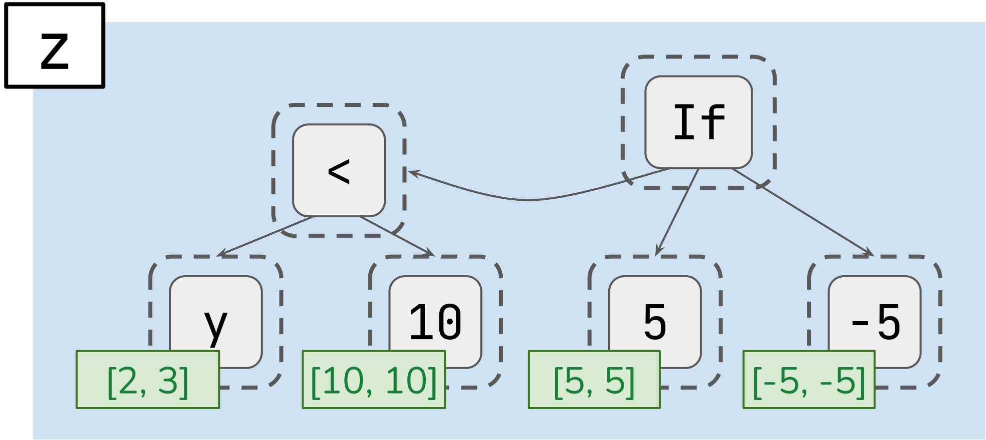
```
(rule ((= e (If pred then else))
       (= then-ival (ival then))
       (= else-ival (ival else)))
  ((set (ival e) (interval-union then-ival else-ival))))
```

```
if x > 0:
  y = 2
else:
  y = 3
```



Optimization: Interval Analysis

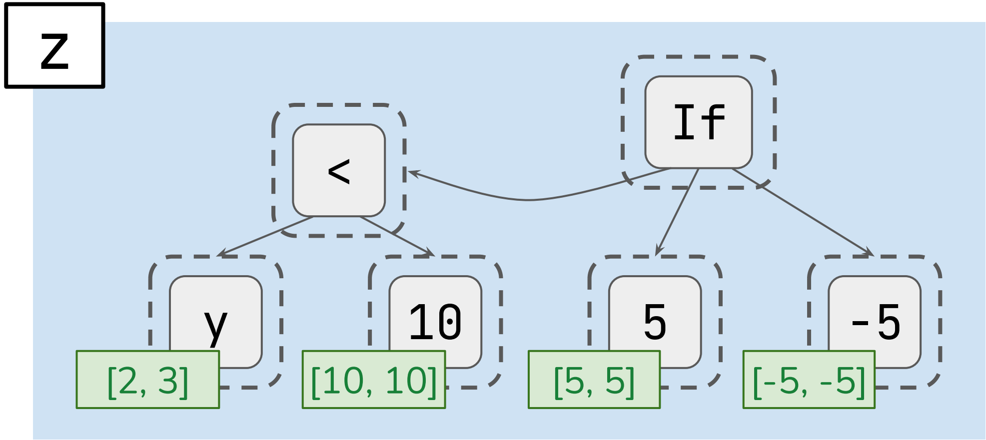
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule ((= e (LessThan x y))
       (= (IntI lo-x hi-x) (ival x))
       (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```

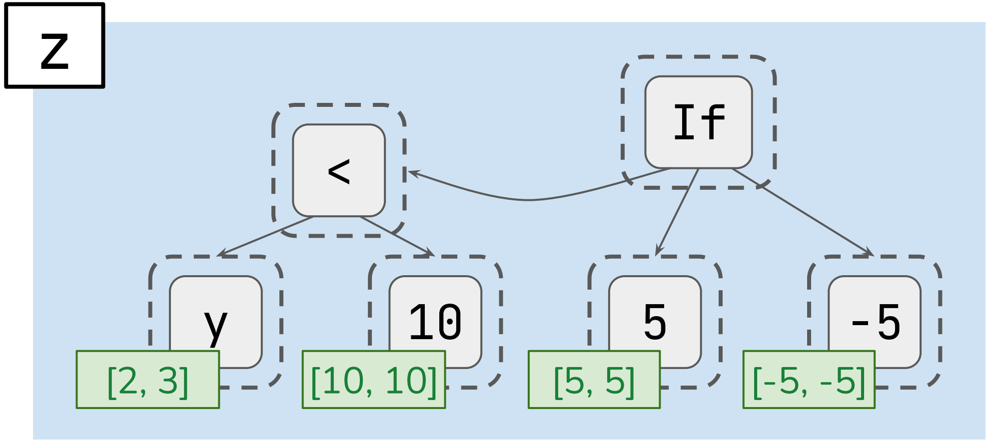
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule ((= e (LessThan x y))
       (= (IntI lo-x hi-x) (ival x))
       (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```

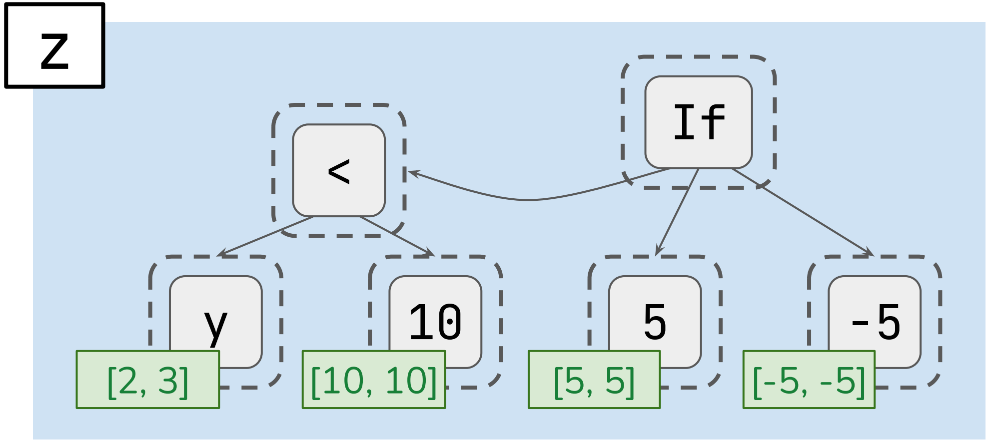
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule ((= e (LessThan x y))
       (= (IntI lo-x hi-x) (ival x))
       (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```

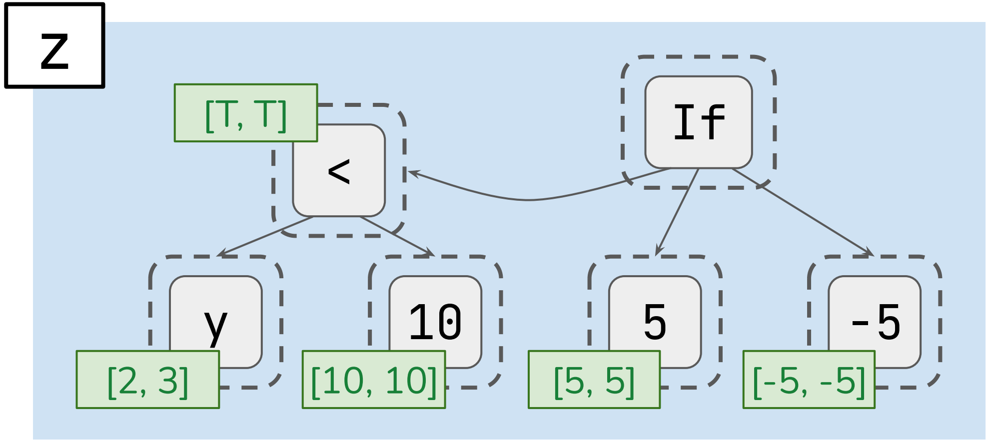
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule ((= e (LessThan x y))
       (= (IntI lo-x hi-x) (ival x))
       (= (IntI lo-y hi-y) (ival y)))
  ((set (ival e) (BoolI (< hi-x lo-y) (< lo-x hi-y)))))
```

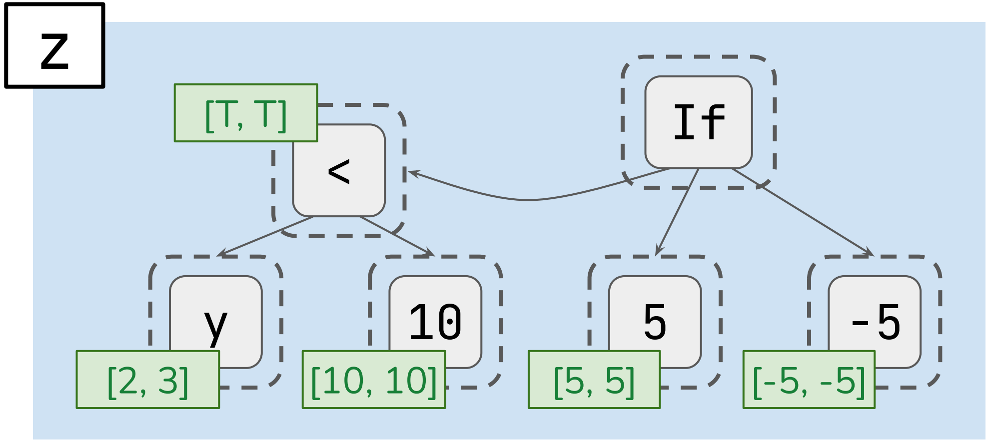
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule
  ((= e (If pred then else))
   (= (BoolI true true) (ival pred)))
  ((union e then)))
```

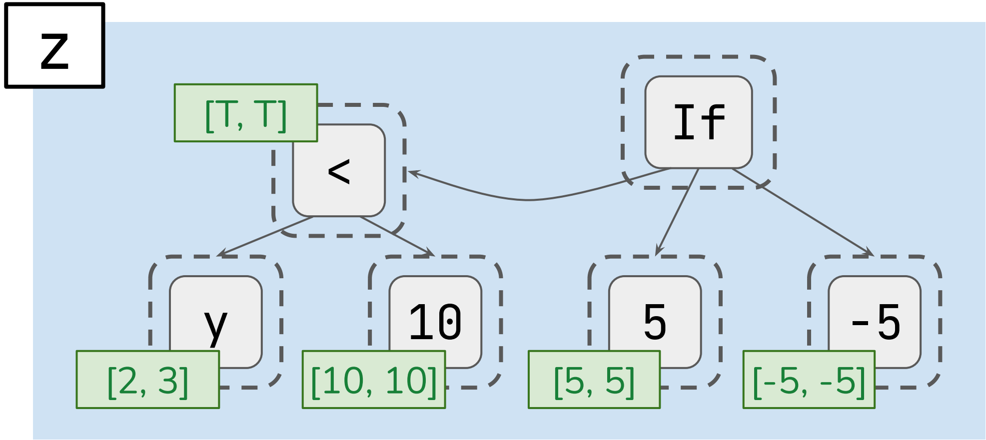
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule
  ((= e (If pred then else))
   (= (BoolI true true) (ival pred)))
  ((union e then)))
```

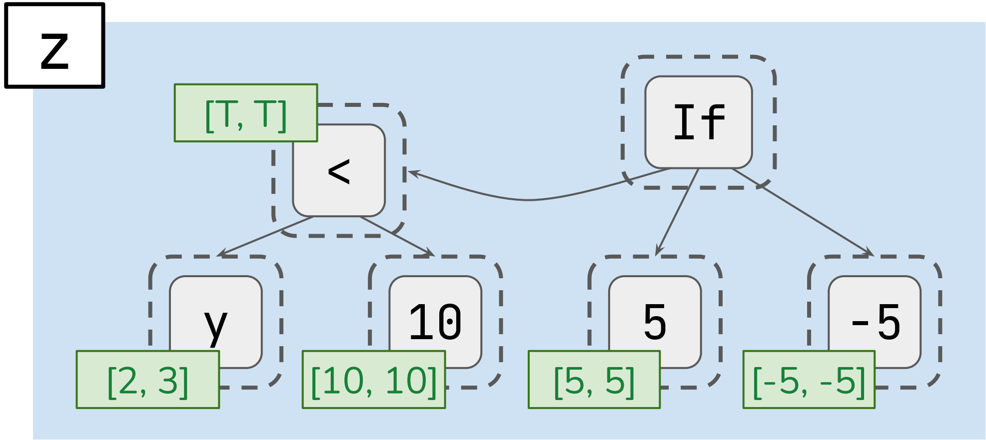
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

```
(rule
  ((= e (If pred then else))
   (= (BoolI true true) (ival pred)))
  ((union e then)))
```

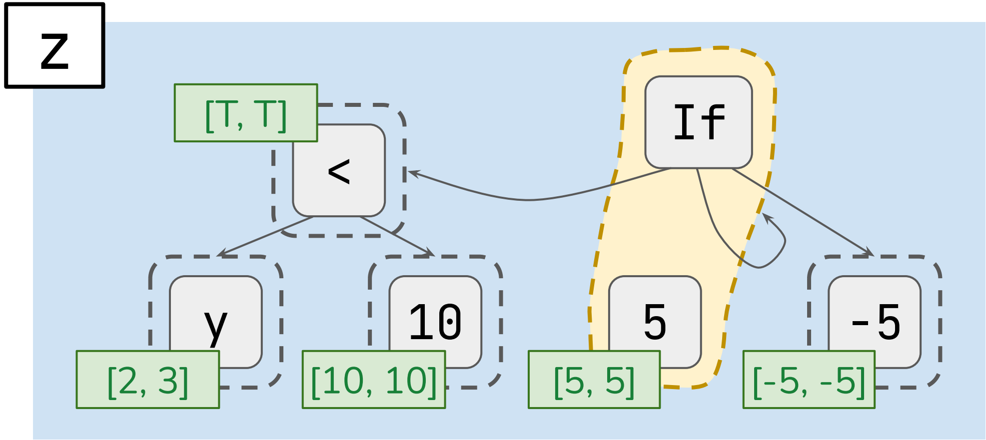
```
if y < 10:
  z = 5
else:
  z = -5
```



Optimization: Interval Analysis

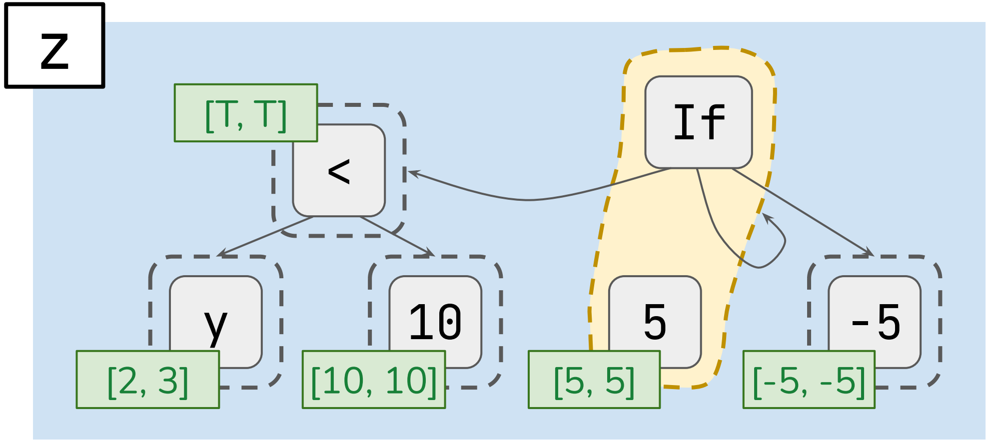
```
(rule
  ((= e (If pred then else))
   (= (BoolI true true) (ival pred)))
  ((union e then)))
```

```
if y < 10:
  z = 5
else:
  z = -5
```

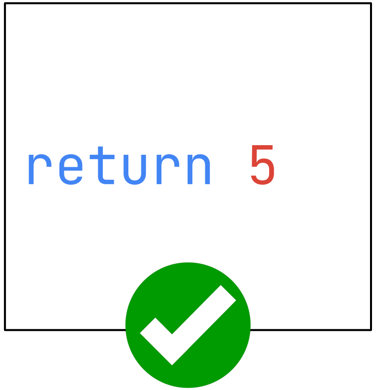


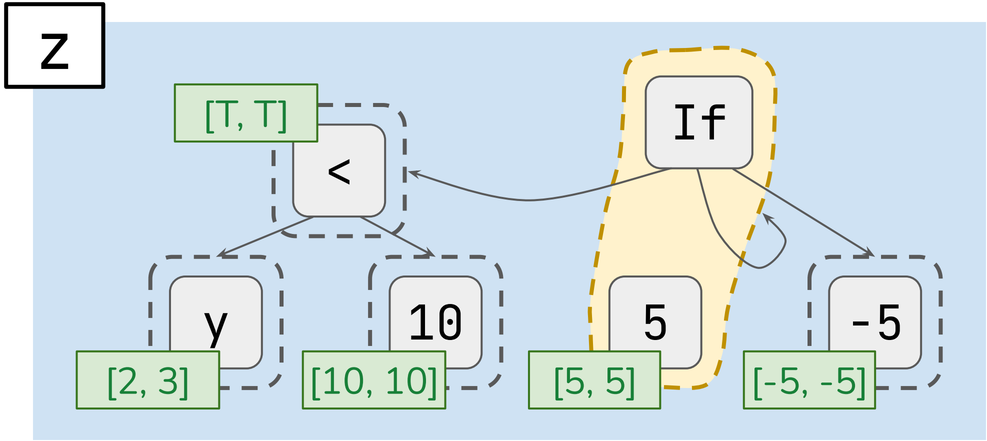
Optimization: Interval Analysis

```
return z
```



Optimization: Interval Analysis





```
if x > 0:
  y = 2
else:
  y = 3
if y < 10:
  z = 5
else:
  z = -5
return z
```

```
return 5
```

What about imperative code?

```
def foo():
  print "hi"
  return 3

foo() + foo()
```

```
def foo():
  print "hi"
  return 3

2 * foo()
```

Are these programs equivalent?

https://ajpal.github.io/

eggcc: An optimizing compiler built with Egglog

- Building a compiler is hard
  - Optimizations don't compose well
  - Incremental analysis is hard
  - Phase ordering gets in the way
- Building a compiler using egglog is better
  - Optimizations are written as declarative rewrite rules
  - Leverage composable analyses from Datalog with fast equational reasoning from EqSat
- There are still challenges
  - Encoding control flow and effectful programs is difficult
  - Tension between sharing and context
  - Extracting the optimal terms from the e-graph after optimization is hard

All of this work is still in progress!

I'd love to talk about it in more detail if you're interested!
Get in touch: anjalip@uw.edu

PLSE

eggcc team