Fast and Flexible Program Optimization with E-Graphs

Anjali Pal

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

PLSE

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + abs(x) - x
  z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + abs(x) - x
  z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + 0
  z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + 0
  z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
 z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
 z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
 z = foo(x, y)
else:
  y = y + 0
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
 z = foo(x, y)
else:
  y = y + 0
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
 z = foo(x, y)
else:
 z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + abs(x)
  z = foo(x, y)
else:
  y = y + abs(x)
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
    y = y + abs(x)
    z = foo(x, y)
else:
    y = y + abs(x)
    z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
y = y + abs(x)
if x > 0:
 z = foo(x, y)
else:
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 3
if x > 0:
 z = foo(x, y)
else:
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + abs(x) - x
  z = foo(x, y)
else:
  y = y + abs(x) + x
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 2
if x > 0:
  z = foo(x, y)
else:
  z = bar(x, y)
```

Most compilers can’t do both!

```
x = input()
y = abs(x) * 2
if x > 0:
  y = y + abs(x)
  z = foo(x, y)
else:
  y = y + abs(x)
  z = bar(x, y)
```

```
x = input()
y = abs(x) * 3
if x > 0:
  z = foo(x, y)
else:
  z = bar(x, y)
```

(I) Term Rewriting

(II) Non-destructive term rewriting using e-graphs

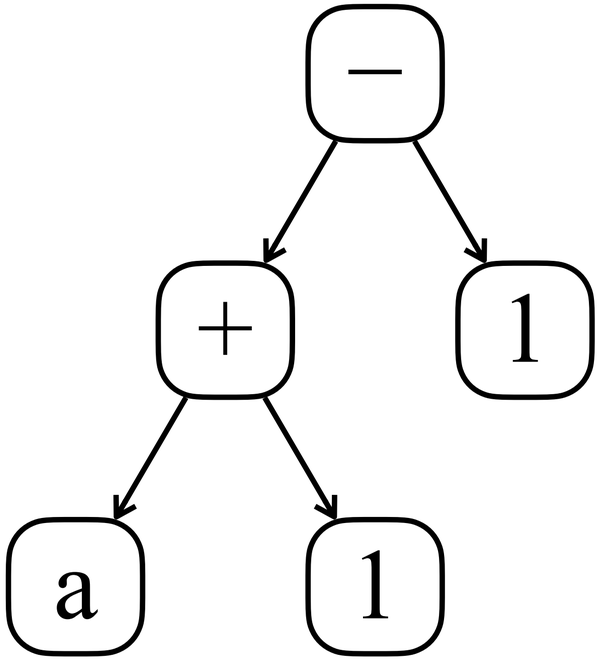
(III) Extracting optimized terms from e-graphs

(IV) Effectful program optimization using e-graphs

(V) Effect-safe extraction

Program Optimization via Term Rewriting

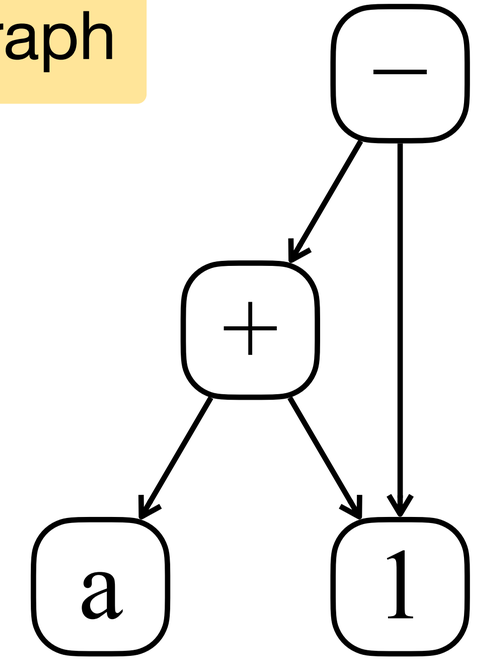
Programs are trees




```
(a + 1) - 1
```

Programs are ~~trees~~ DAGs

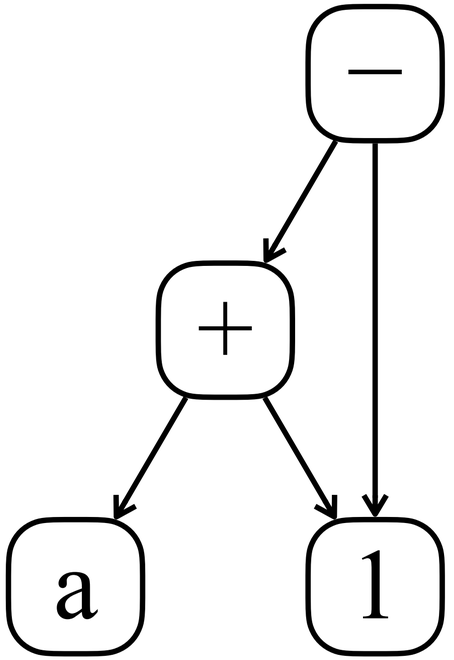
Directed Acyclic Graph




```
(a + 1) - 1
```

Syntactic Rewrite Rules

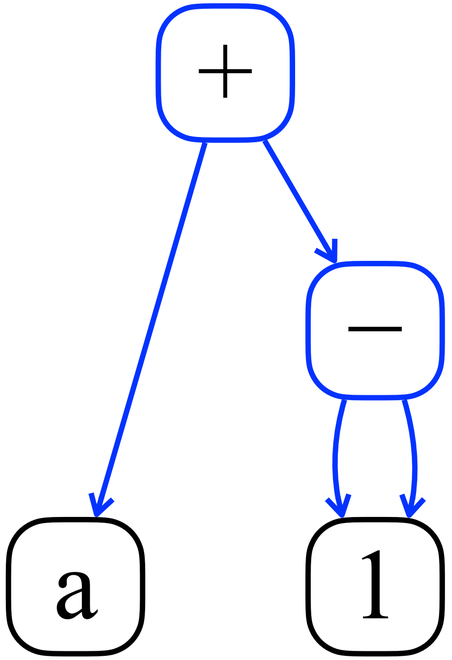
$$(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$$




```
(a + 1) - 1
```

Syntactic Rewrite Rules

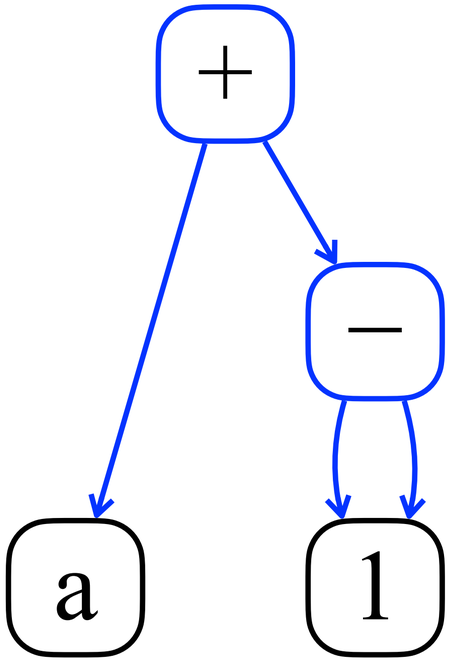
$$(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$$




```
a + (1 - 1)
```

Syntactic Rewrite Rules

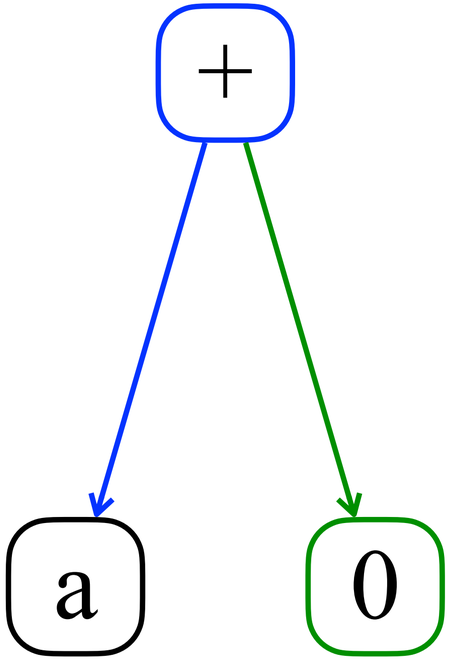
$$\alpha - \alpha \rightsquigarrow 0$$




```
a + (1 - 1)
```

Syntactic Rewrite Rules

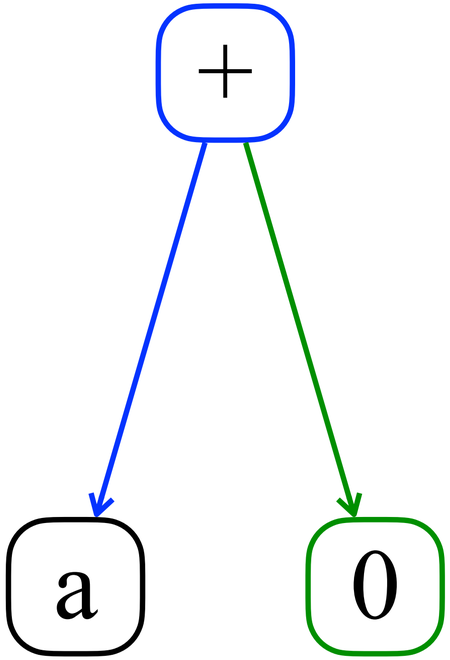
$$\alpha - \alpha \rightsquigarrow 0$$




```
a + 0
```

Syntactic Rewrite Rules

$$\alpha + 0 \rightsquigarrow \alpha$$



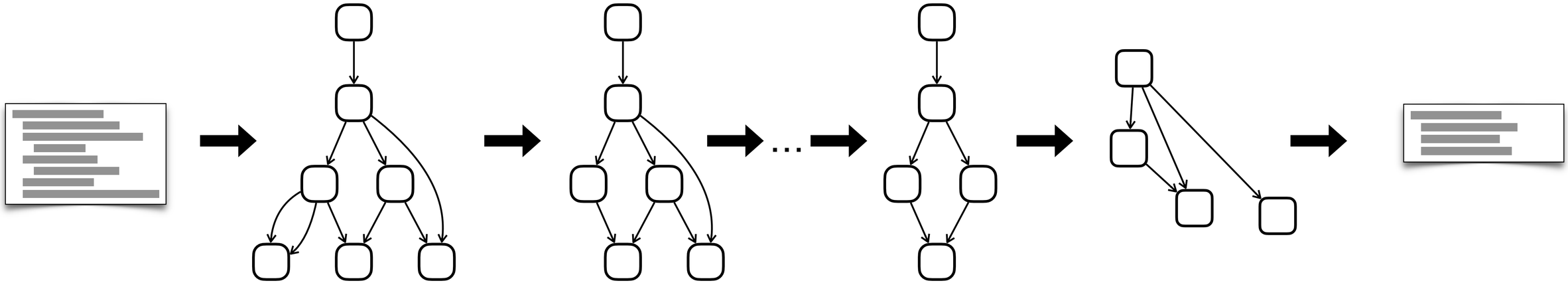

```
a + 0
```

Syntactic Rewrite Rules

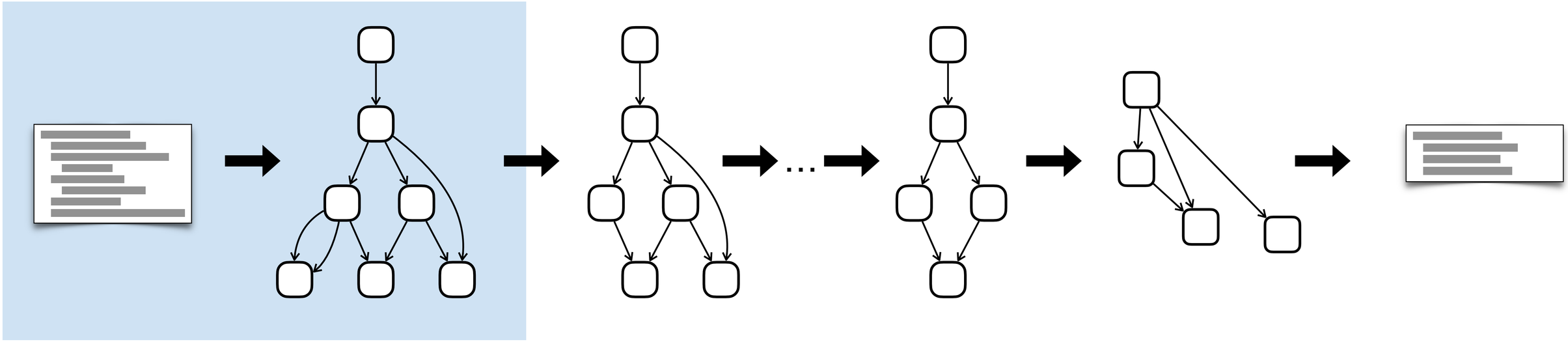
$$\alpha + 0 \rightsquigarrow \alpha$$

a

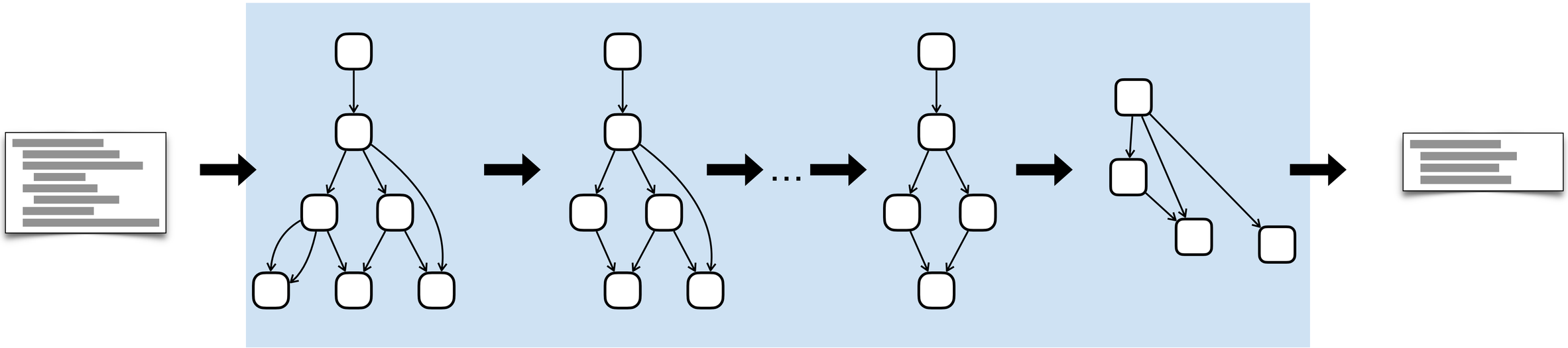
Program Optimization with Term Rewriting



Program Optimization with Term Rewriting

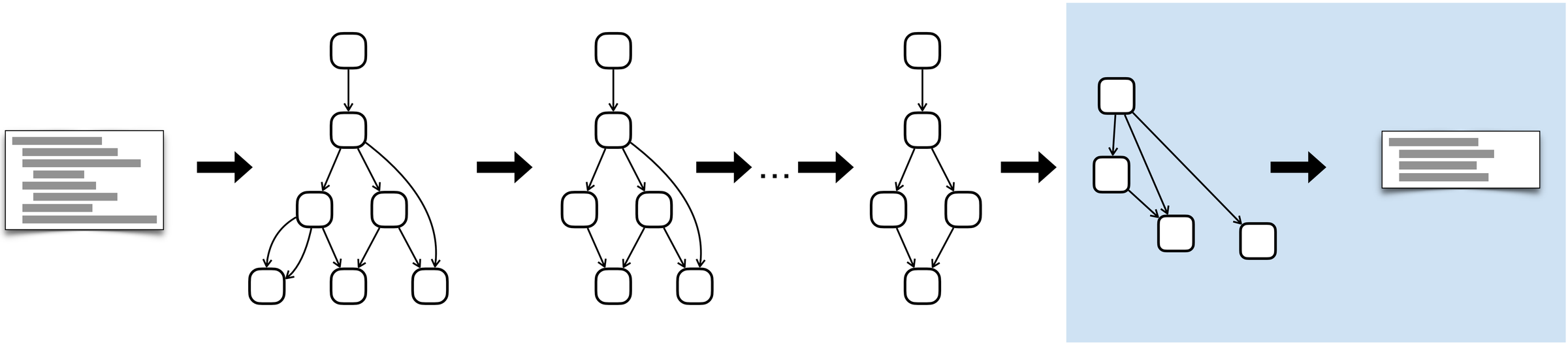


Program Optimization with Term Rewriting



Fixed Rule Order

Program Optimization with Term Rewriting

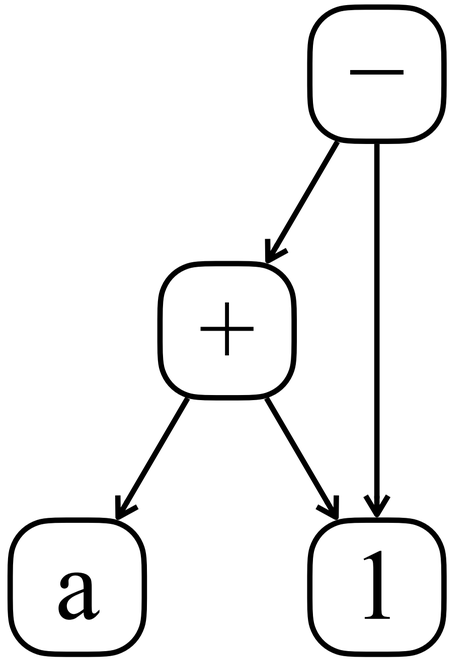


Phase Ordering Problem

1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$

2. $\alpha - \alpha \rightsquigarrow 0$

3. $\alpha + 0 \rightsquigarrow \alpha$



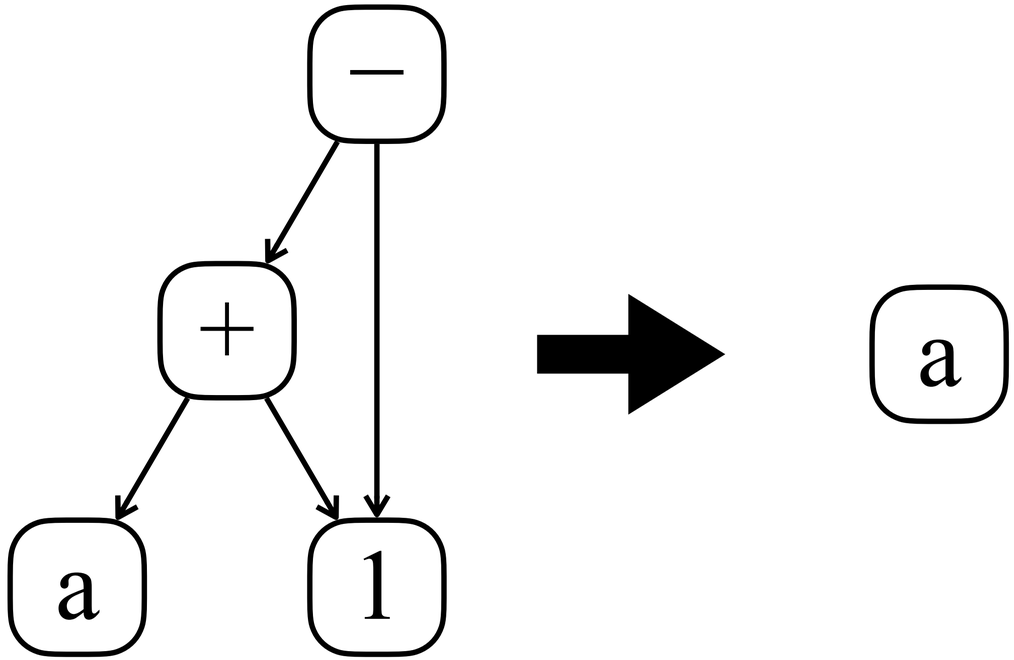

```
(a + 1) - 1
```

Phase Ordering Problem

1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$

2. $\alpha - \alpha \rightsquigarrow 0$

3. $\alpha + 0 \rightsquigarrow \alpha$



`(a + 1) - 1`

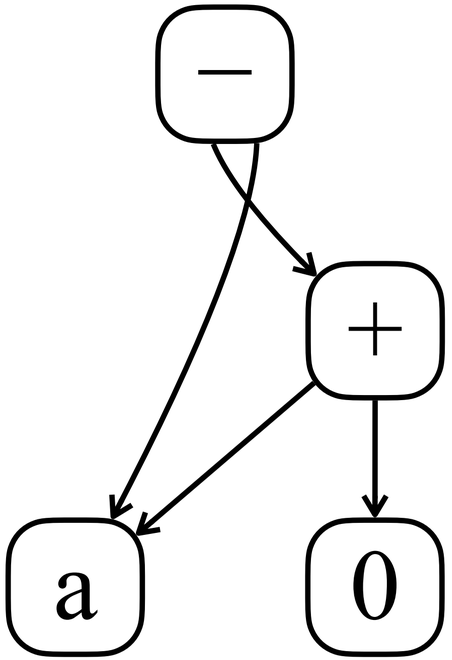
a ✅

Phase Ordering Problem

1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$

2. $\alpha - \alpha \rightsquigarrow 0$

3. $\alpha + 0 \rightsquigarrow \alpha$



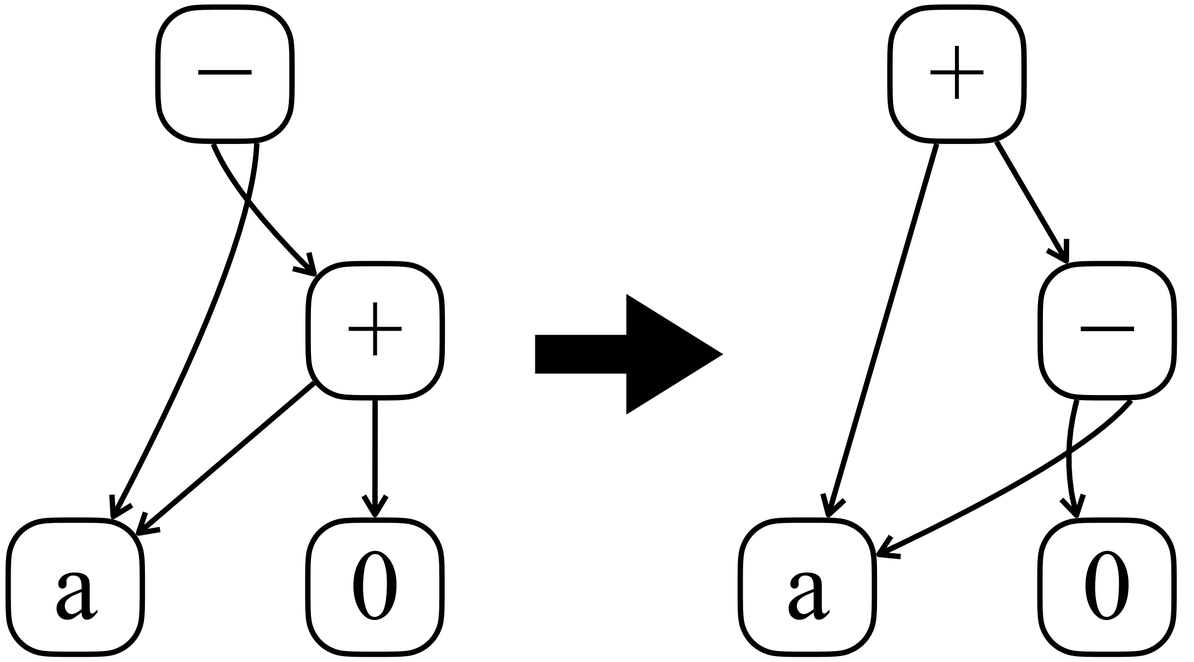

```
(a + 0) - a
```

Phase Ordering Problem

1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$

2. $\alpha - \alpha \rightsquigarrow 0$

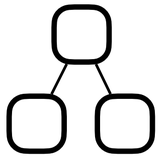
3. $\alpha + 0 \rightsquigarrow \alpha$



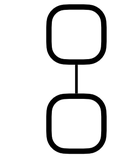
`(a + 0) - a` `a + (0 -a)`

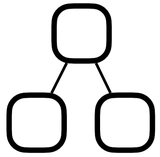
No rule ordering works for all inputs

Try Every Order?

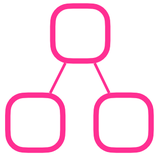


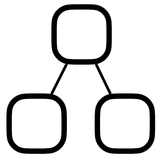
1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$
2. $\alpha - \alpha \rightsquigarrow 0$
3. $\alpha + 0 \rightsquigarrow \alpha$



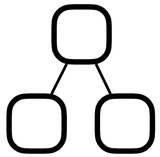


1. $\alpha - \alpha \rightsquigarrow 0$
2. $\alpha + 0 \rightsquigarrow \alpha$
3. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$

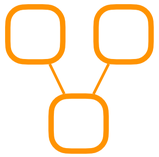


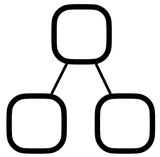


1. $\alpha + 0 \rightsquigarrow \alpha$
2. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$
3. $\alpha - \alpha \rightsquigarrow 0$

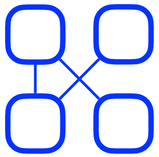


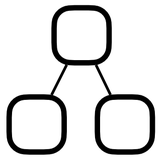
1. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$
2. $\alpha - \alpha \rightsquigarrow 0$
3. $\alpha + 0 \rightsquigarrow \alpha$



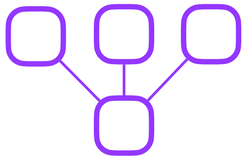


1. $\alpha + 0 \rightsquigarrow \alpha$
2. $\alpha - \alpha \rightsquigarrow 0$
3. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$



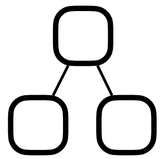


1. $\alpha - \alpha \rightsquigarrow 0$
2. $(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$
3. $\alpha + 0 \rightsquigarrow \alpha$

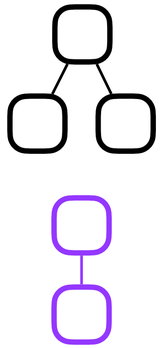


Too much time ❌

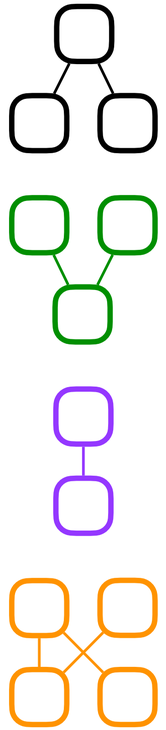
Keep Every Term?



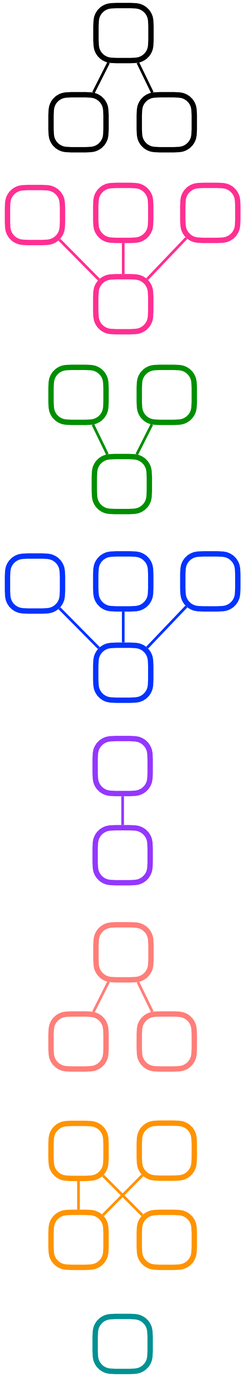
$(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$



$\alpha + 0 \rightsquigarrow \alpha$

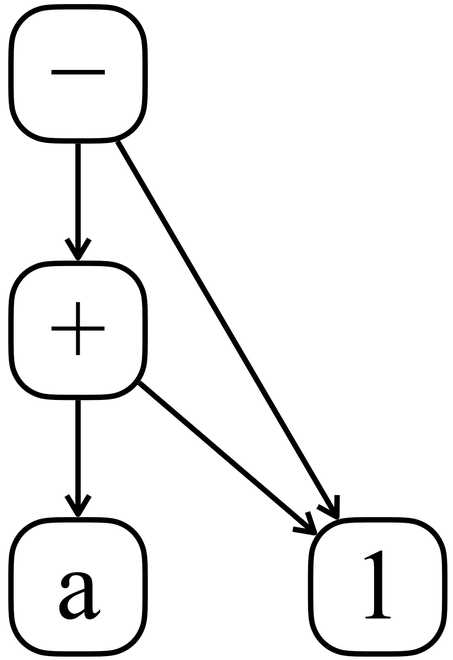


$\alpha - \alpha \rightsquigarrow 0$

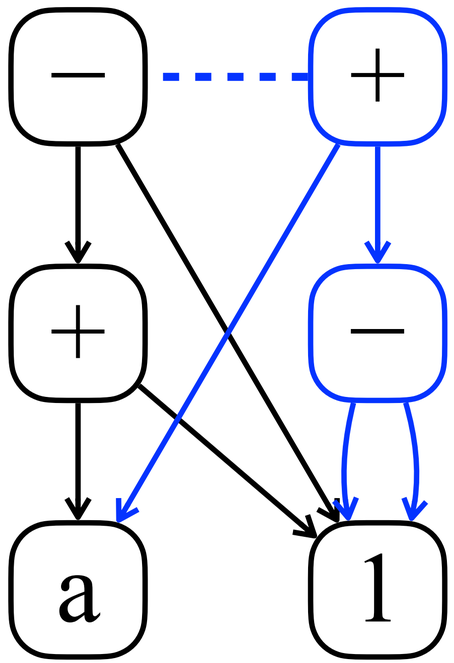


Too much space?

Equivalence Graph (E-Graph)

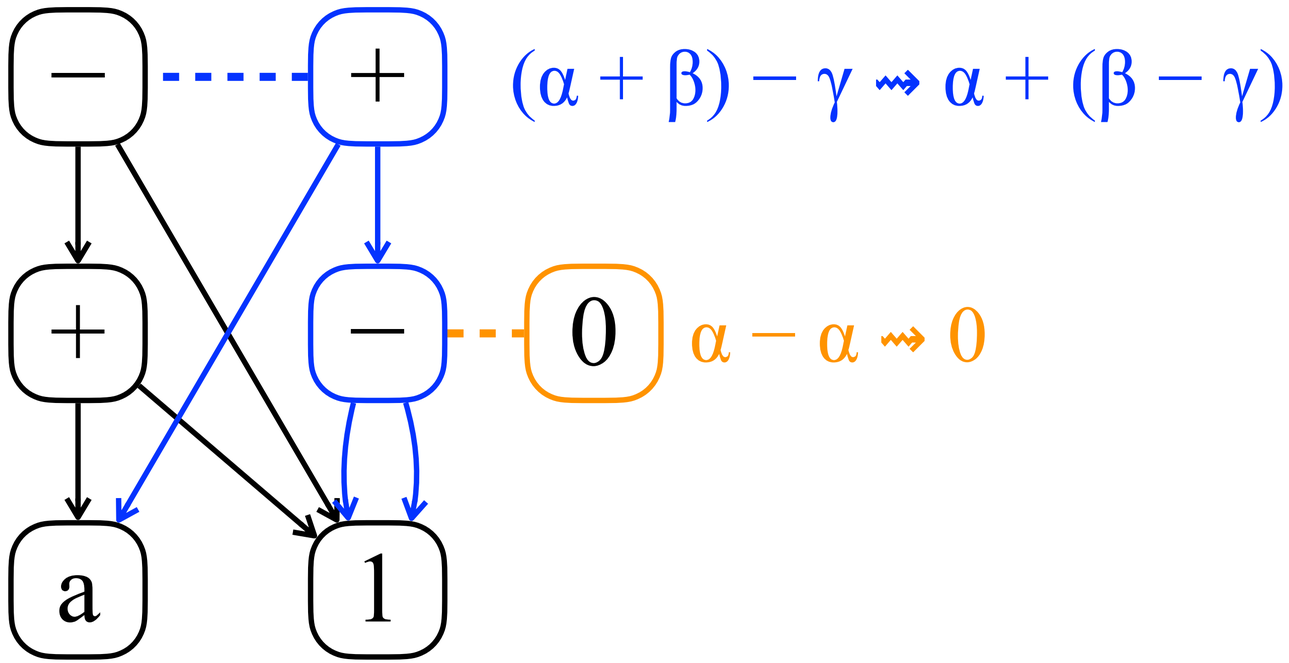


Equivalence Graph (E-Graph)

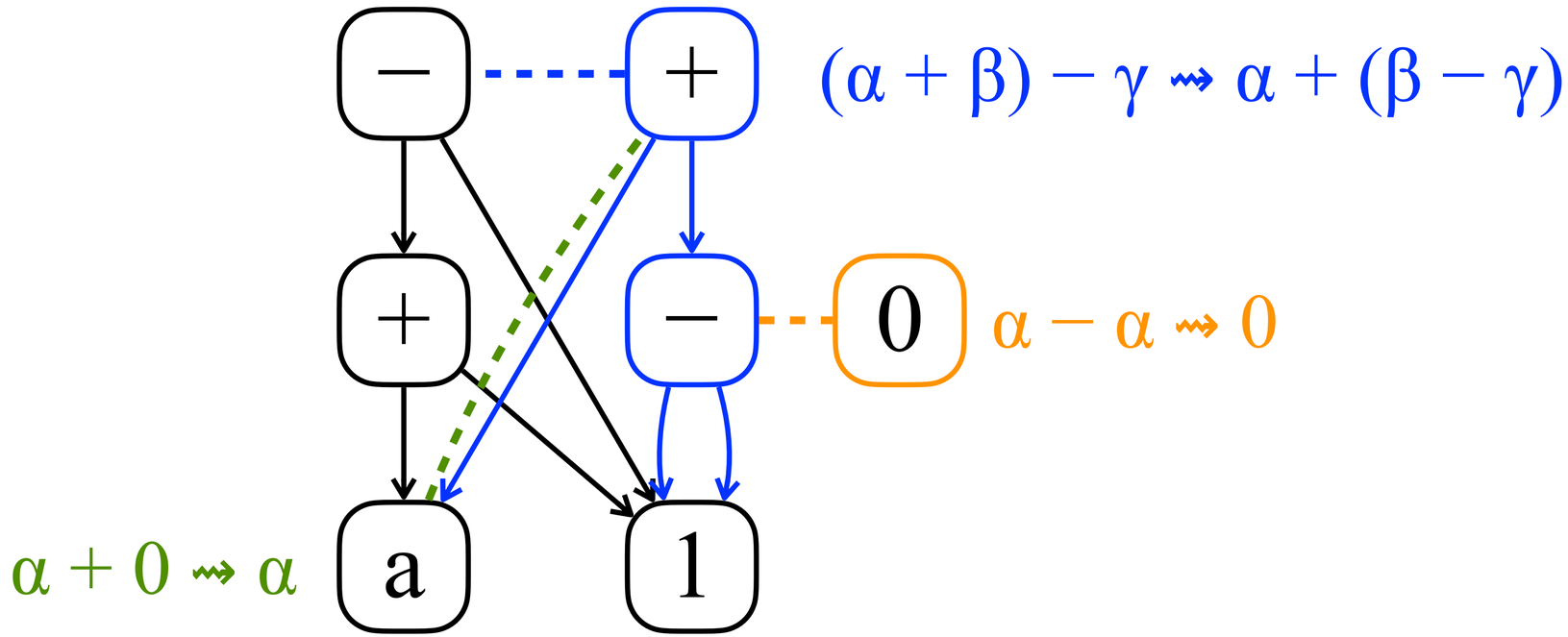


$$(\alpha + \beta) - \gamma \rightsquigarrow \alpha + (\beta - \gamma)$$

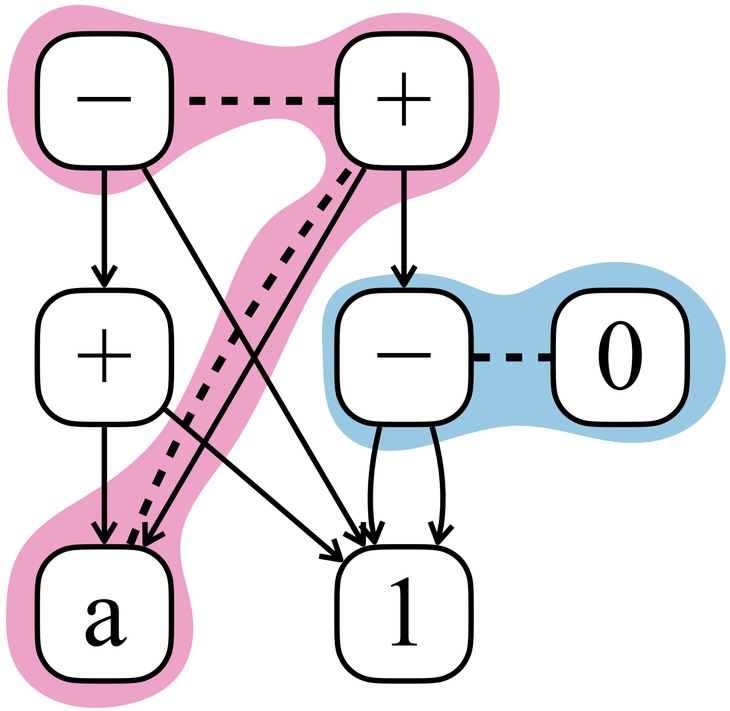
Equivalence Graph (E-Graph)



Equivalence Graph (E-Graph)

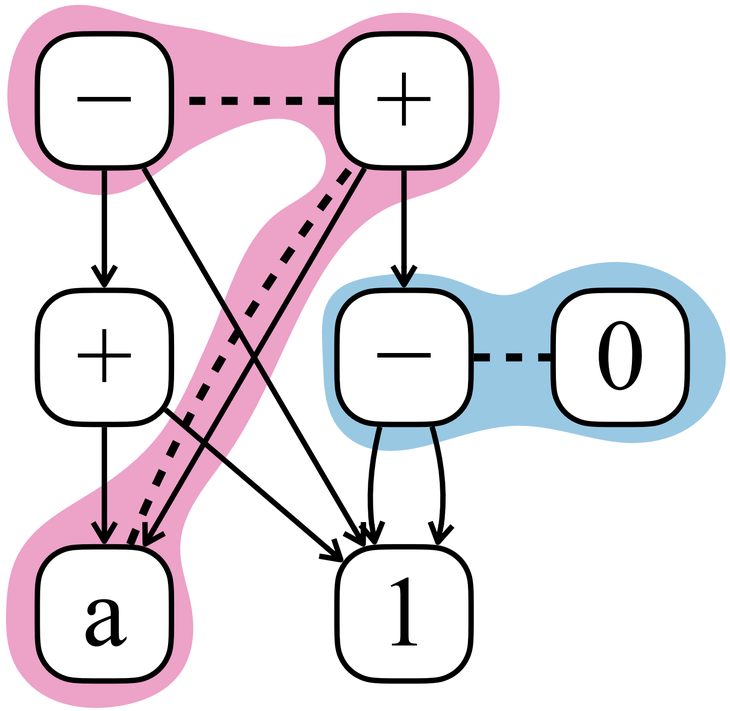


Equivalence Graph (E-Graph)



Equivalence Graph (E-Graph)

$$(a + 1) - 1$$
$$\equiv a + (1 - 1)$$
$$\equiv a$$

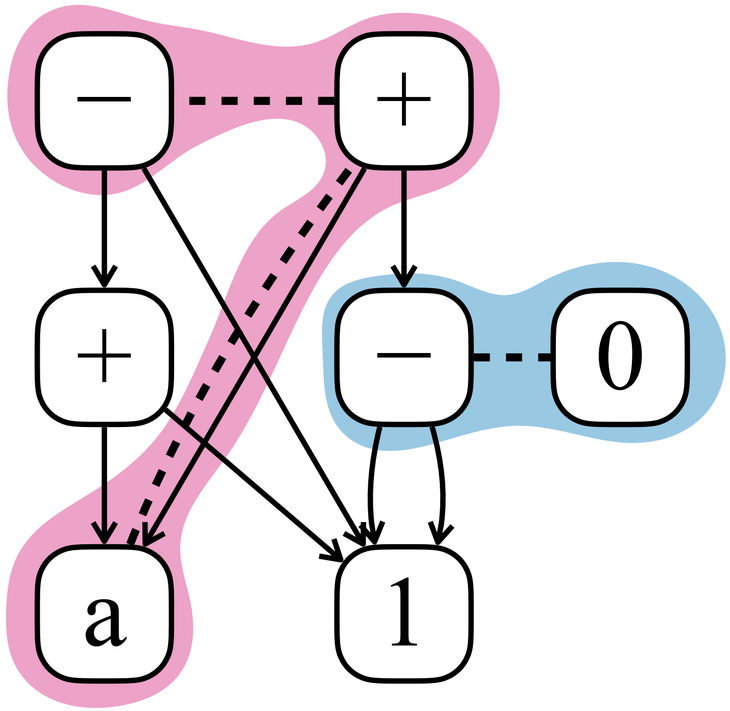


$1 - 1 \equiv 0$

Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

$$(a + 1) - 1$$
$$\equiv a + (1 - 1)$$
$$\equiv a$$

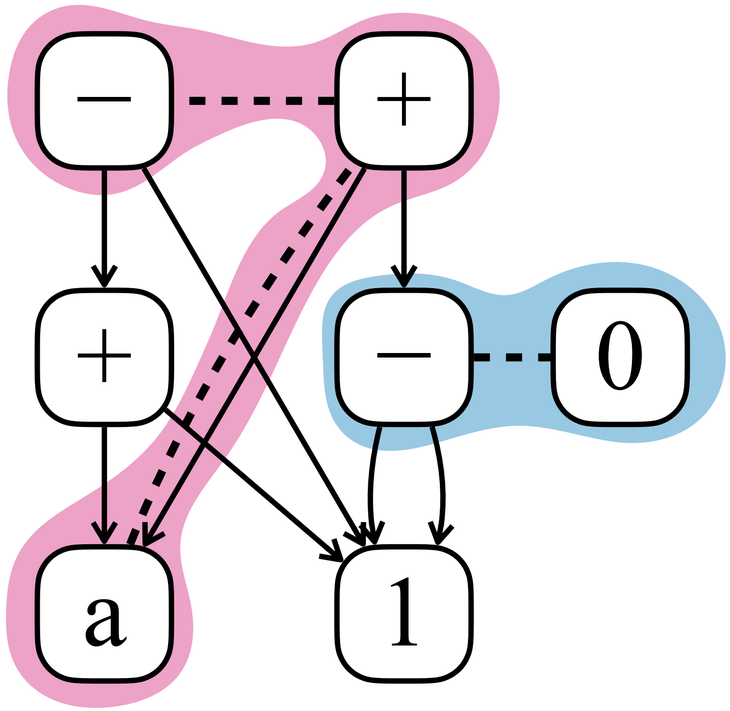


$$1 - 1 \equiv 0$$

Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

$$\begin{aligned}(a + 1) - 1 \\ \equiv a + (1 - 1) \\ \equiv a \\ \equiv a + 0\end{aligned}$$

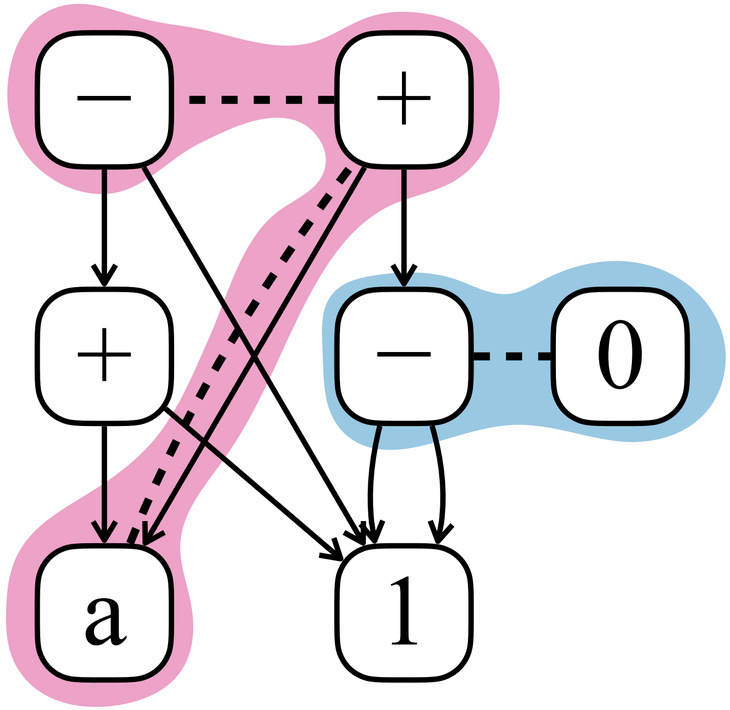


$1 - 1 \equiv 0$

Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

$$
\begin{aligned}
&(a + 1) - 1 \\
&\equiv a + (1 - 1) \\
&\equiv a \\
&\equiv a + 0 \\
&\equiv (a + 0) + 0 \\
&\equiv ((a + 0) + 0) + 0 \\
&\ldots
\end{aligned}
$$



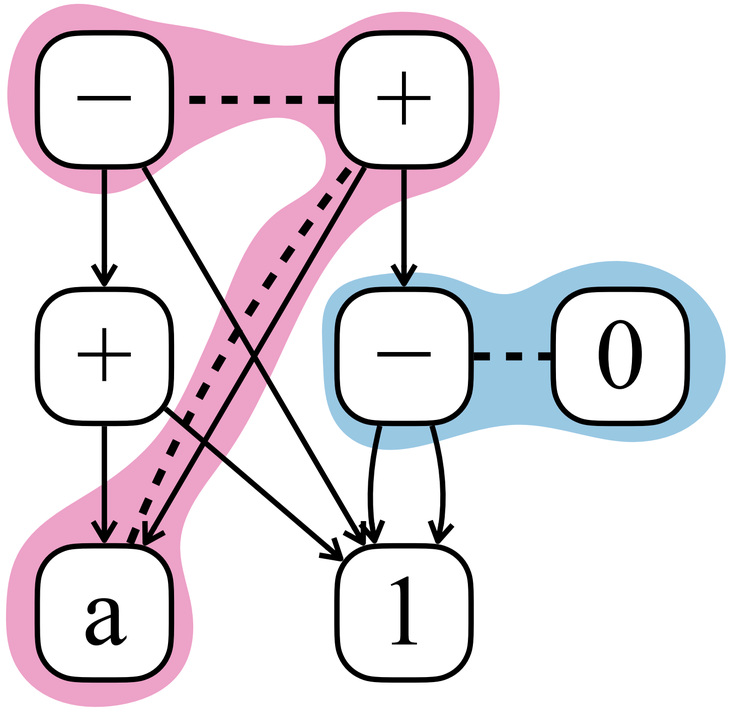
$1 - 1 \equiv 0$

Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

Equivalent terms are interchangeable

$$(a + 1) - 1$$
$$\equiv a + (1 - 1)$$
$$\equiv a$$
$$\equiv a + 0$$
$$\equiv (a + 0) + 0$$
$$\equiv ((a + 0) + 0) + 0$$
$$\ldots$$

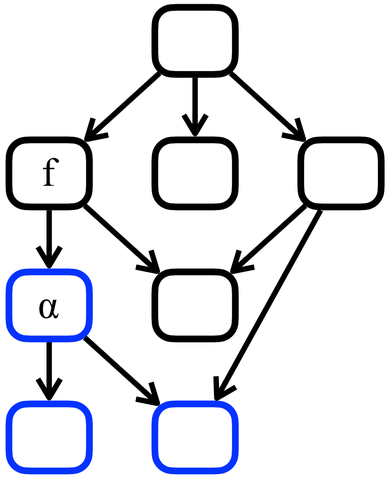


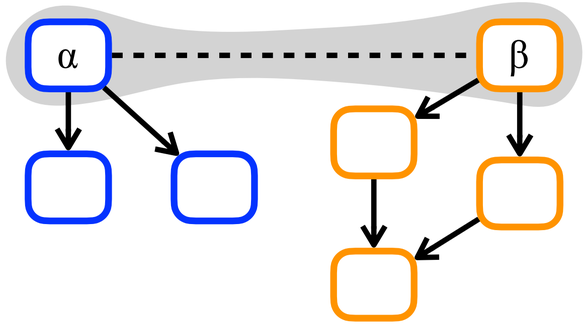
$1 - 1 \equiv 0$

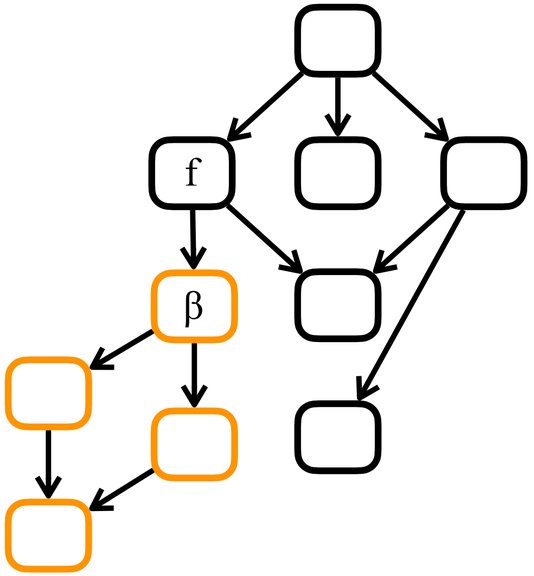
Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

Equivalent terms are interchangeable



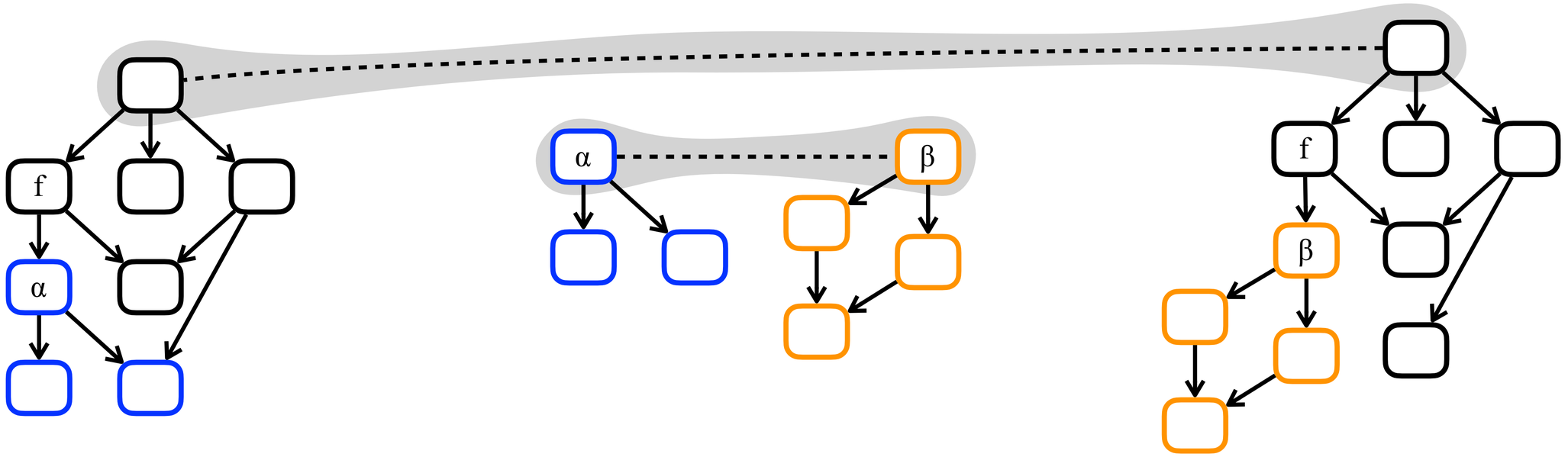




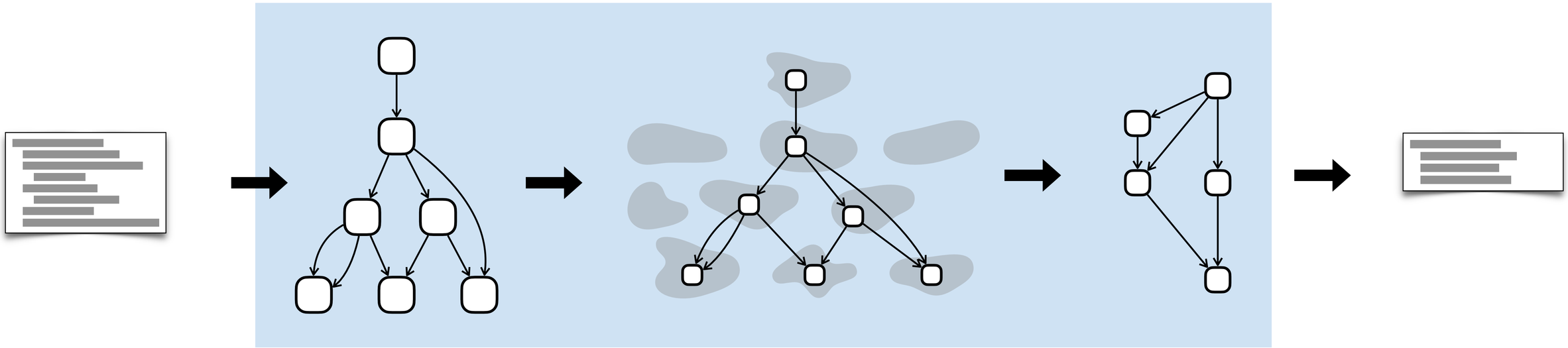
Equivalence Graph (E-Graph)

$$\alpha \equiv \beta \Rightarrow f(\alpha) \equiv f(\beta)$$

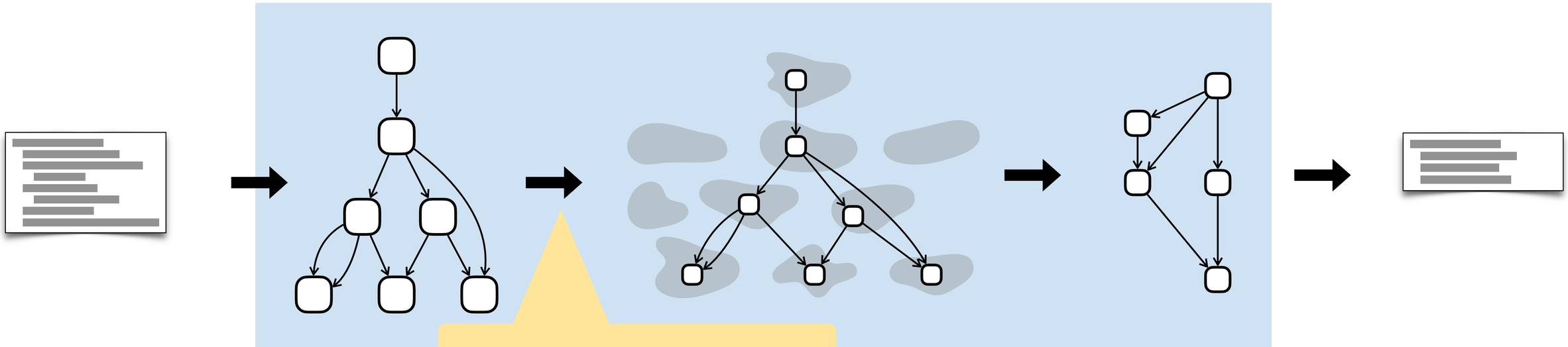
Equivalent terms are interchangeable



Program Optimization with E-Graphs

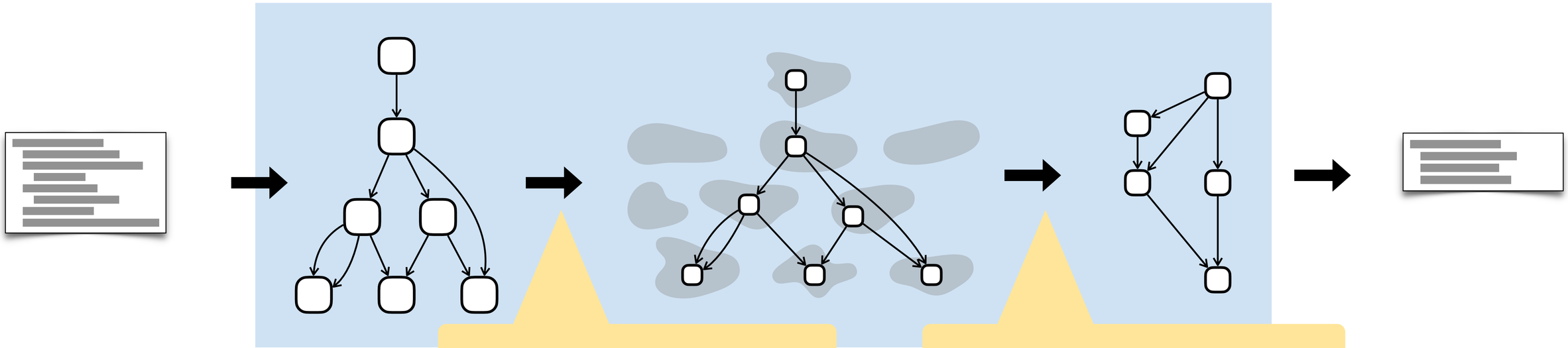


Program Optimization with E-Graphs

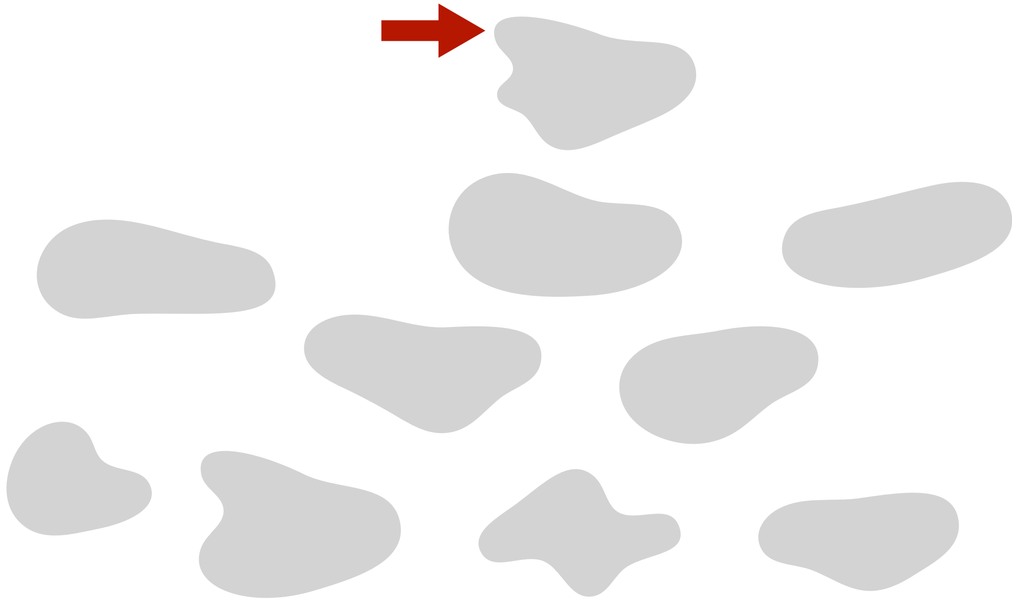


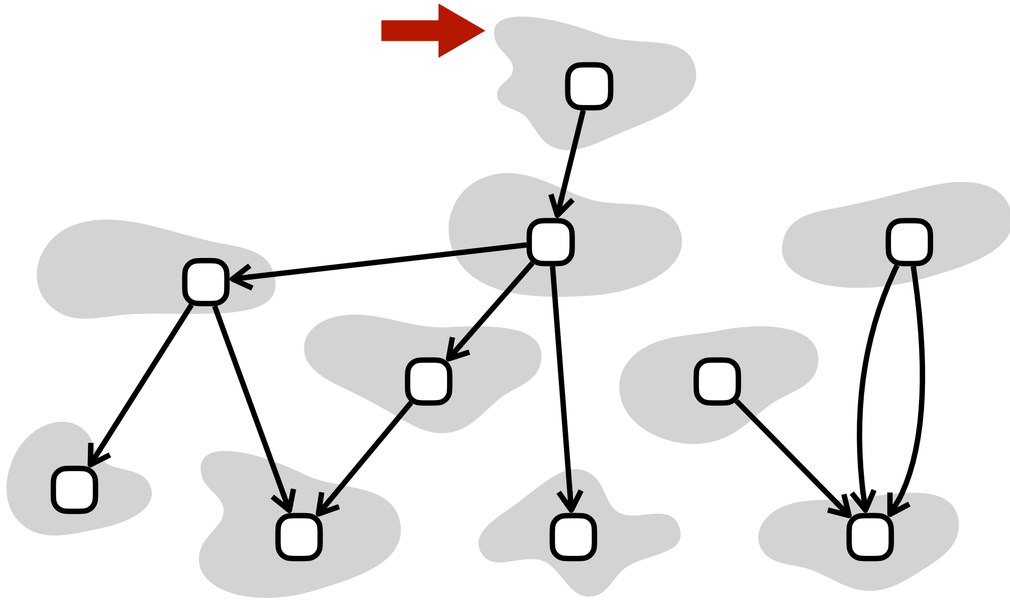
Non-destructive term rewriting

Program Optimization with E-Graphs



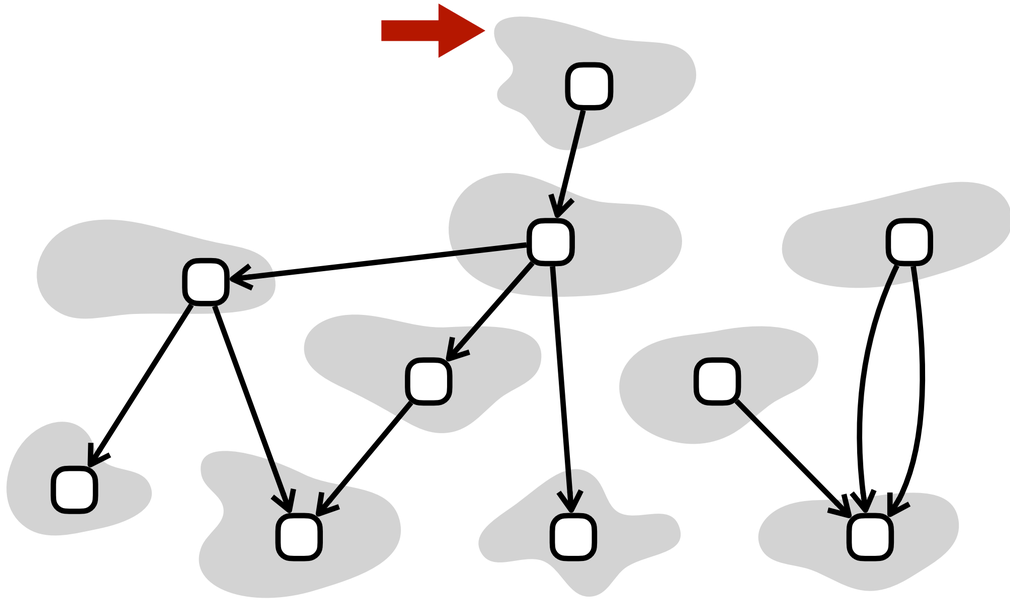
Extraction





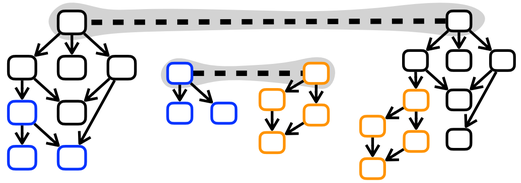
Extraction

1. Pick a representative term from each e-class
2. Re-use that term everywhere you need an equivalent term



Extraction

1. Pick a representative term from each e-class
2. Re-use that term everywhere you need an equivalent term

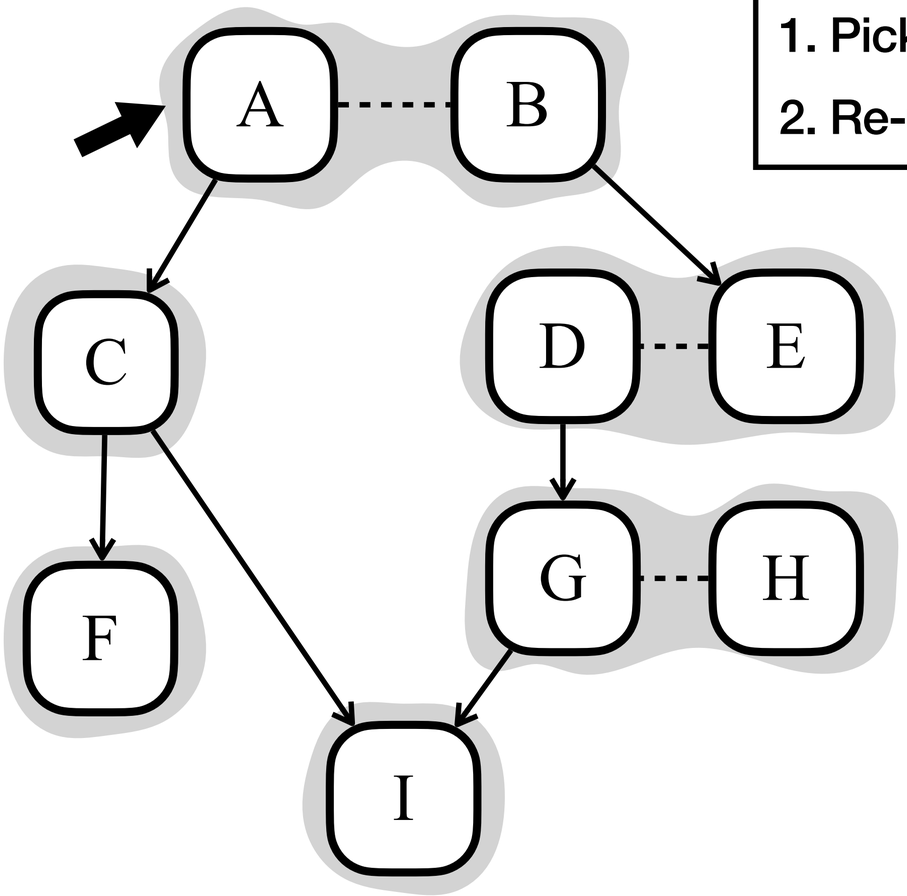


Equivalent terms are interchangeable

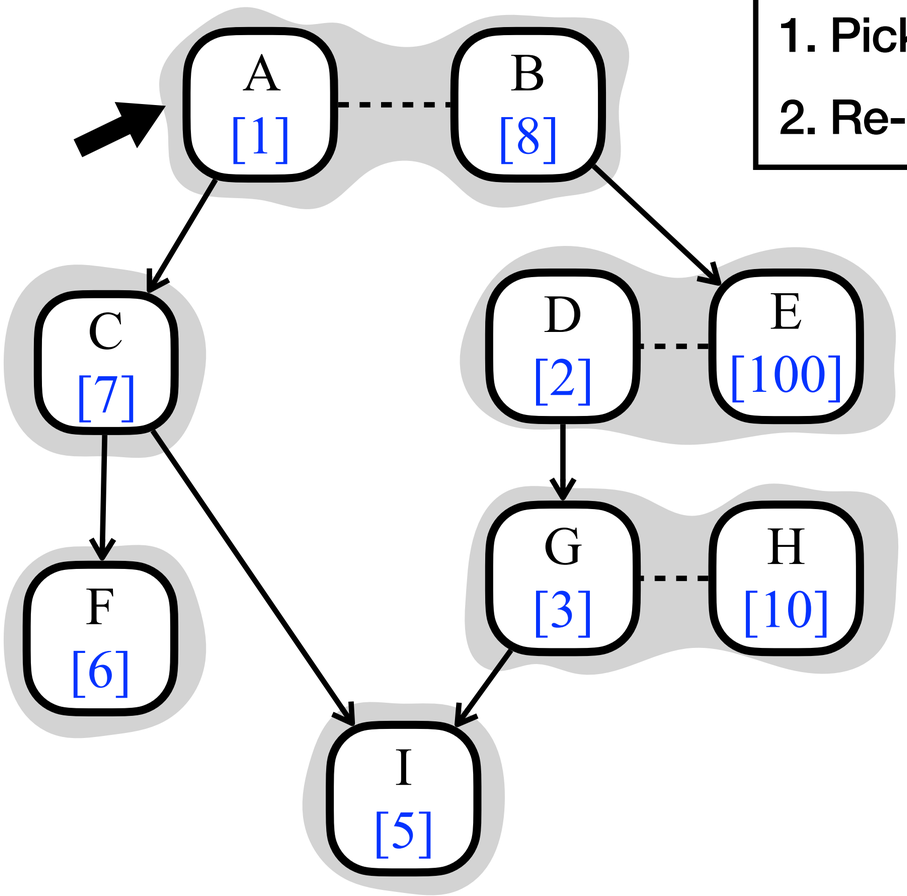
Greedy Extraction with Cost Model

Extraction

1. Pick a term from each e-class
2. Re-use that term everywhere



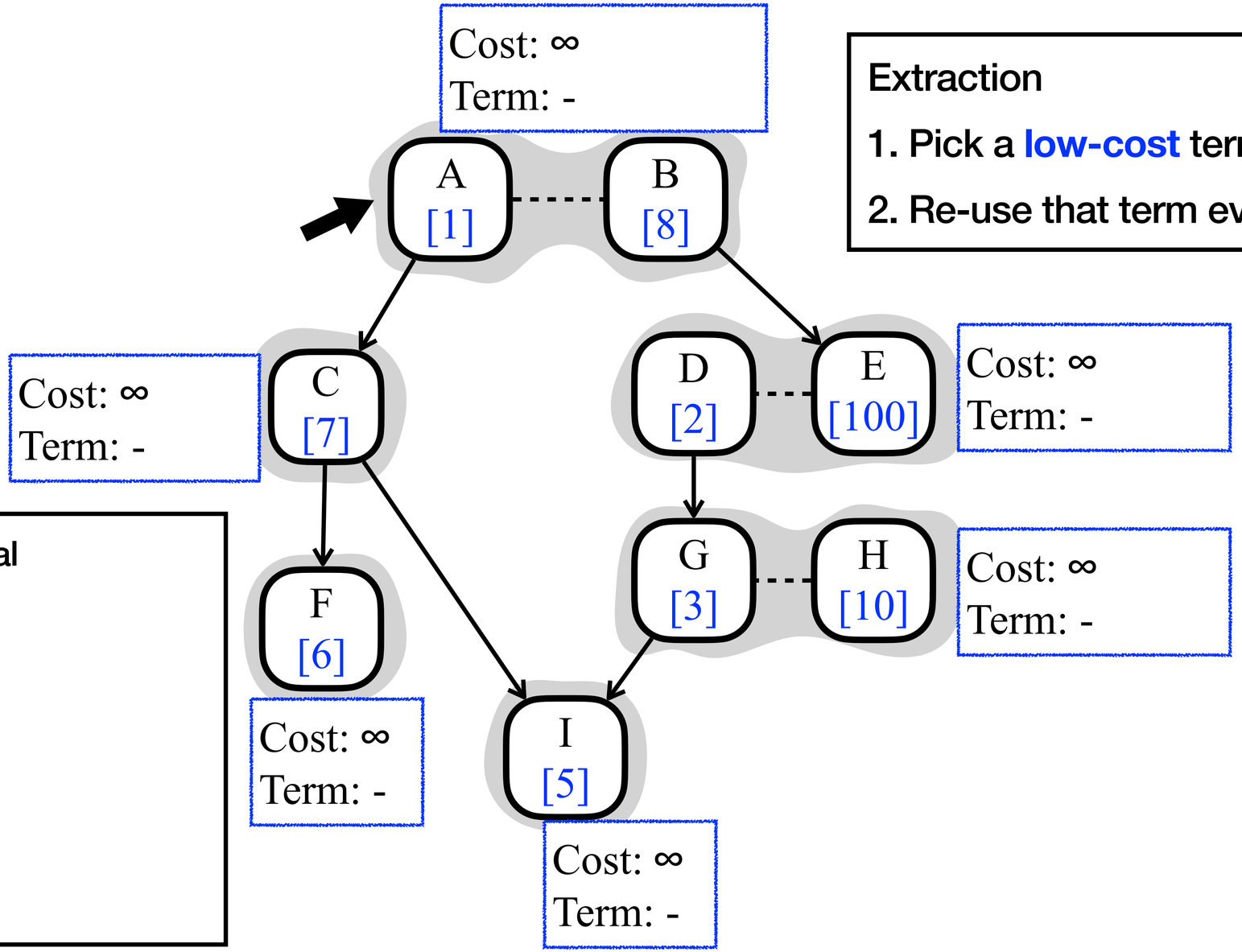
Greedy Extraction with Cost Model



Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Greedy Extraction with Cost Model



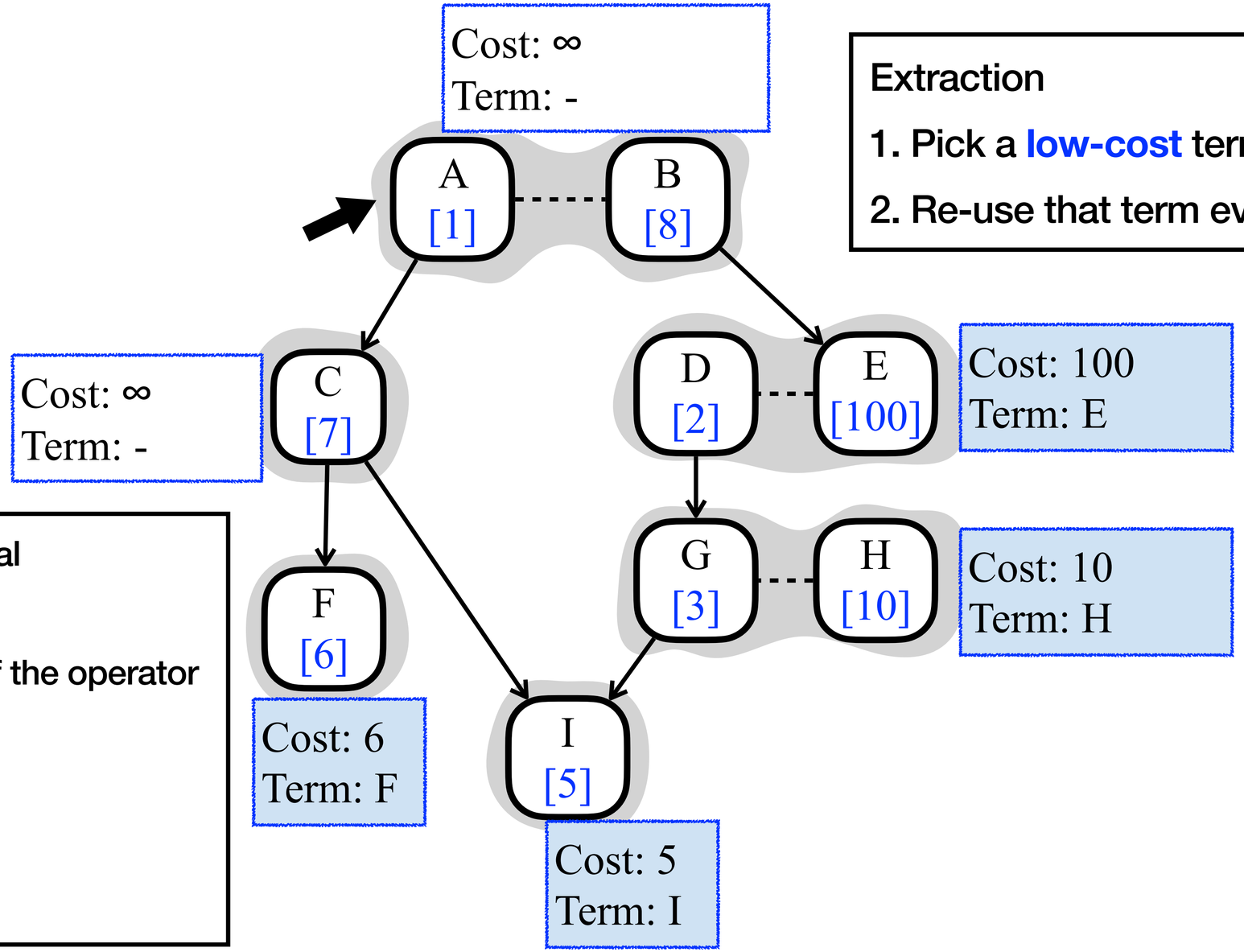
Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Bottom Up Graph Traversal

1. Initialize all costs to ∞

Greedy Extraction with Cost Model



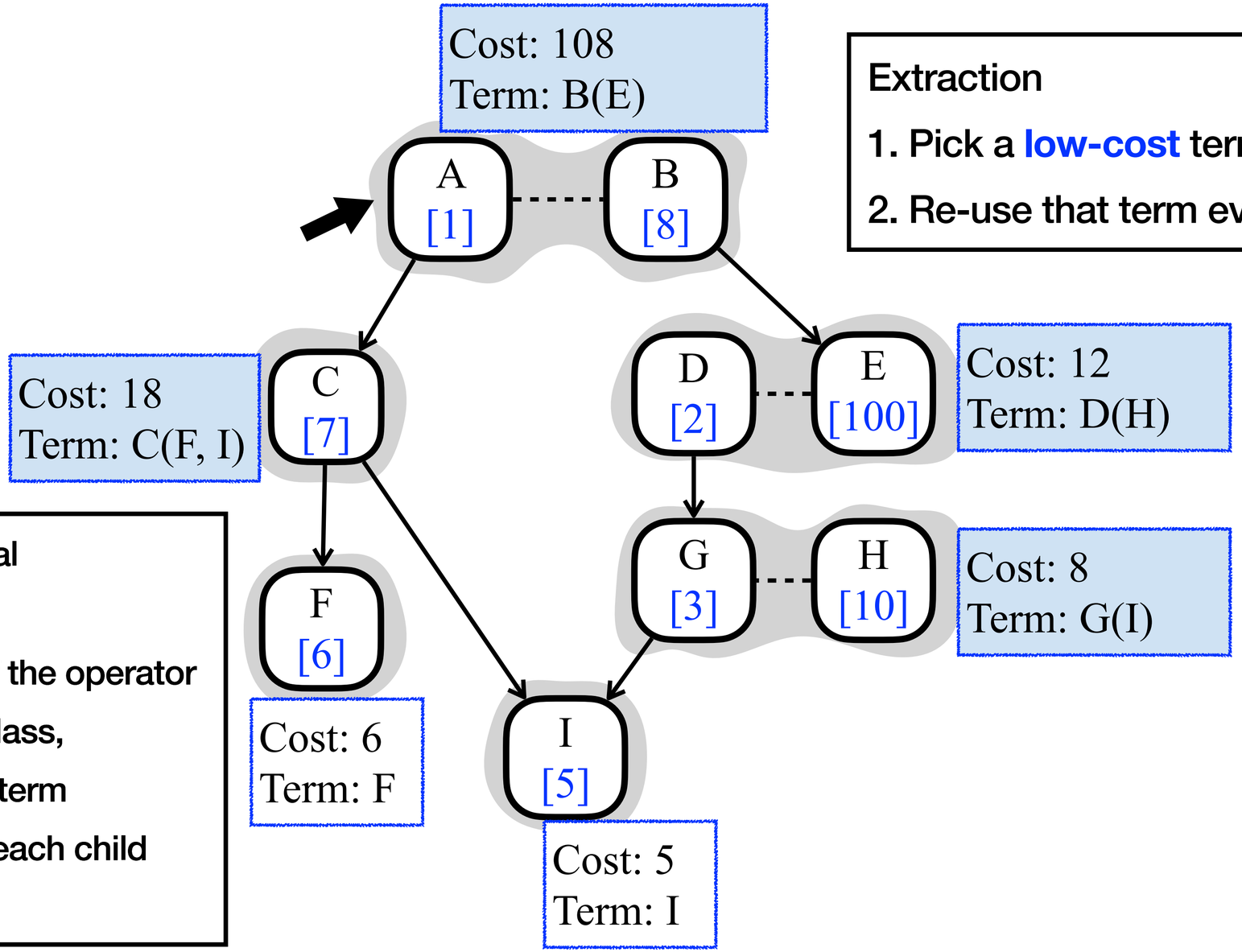
Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Bottom Up Graph Traversal

1. Initialize all costs to ∞
2. Leaf terms have cost of the operator

Greedy Extraction with Cost Model



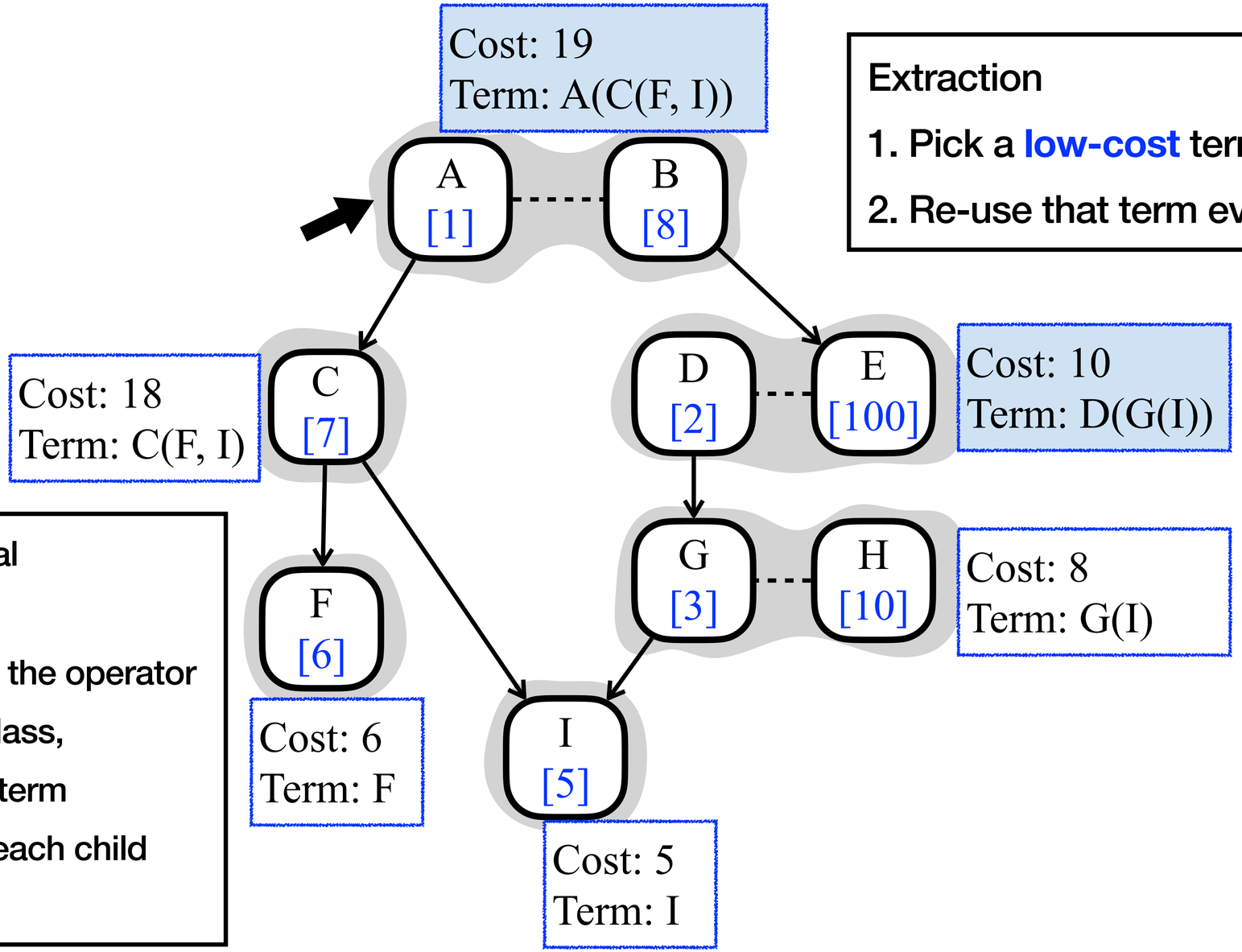
Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Bottom Up Graph Traversal

1. Initialize all costs to ∞
2. Leaf terms have cost of the operator
3. For each equivalence class,
   search for a lower cost term
   using the best cost for each child
4. Repeat until fixed point

Greedy Extraction with Cost Model



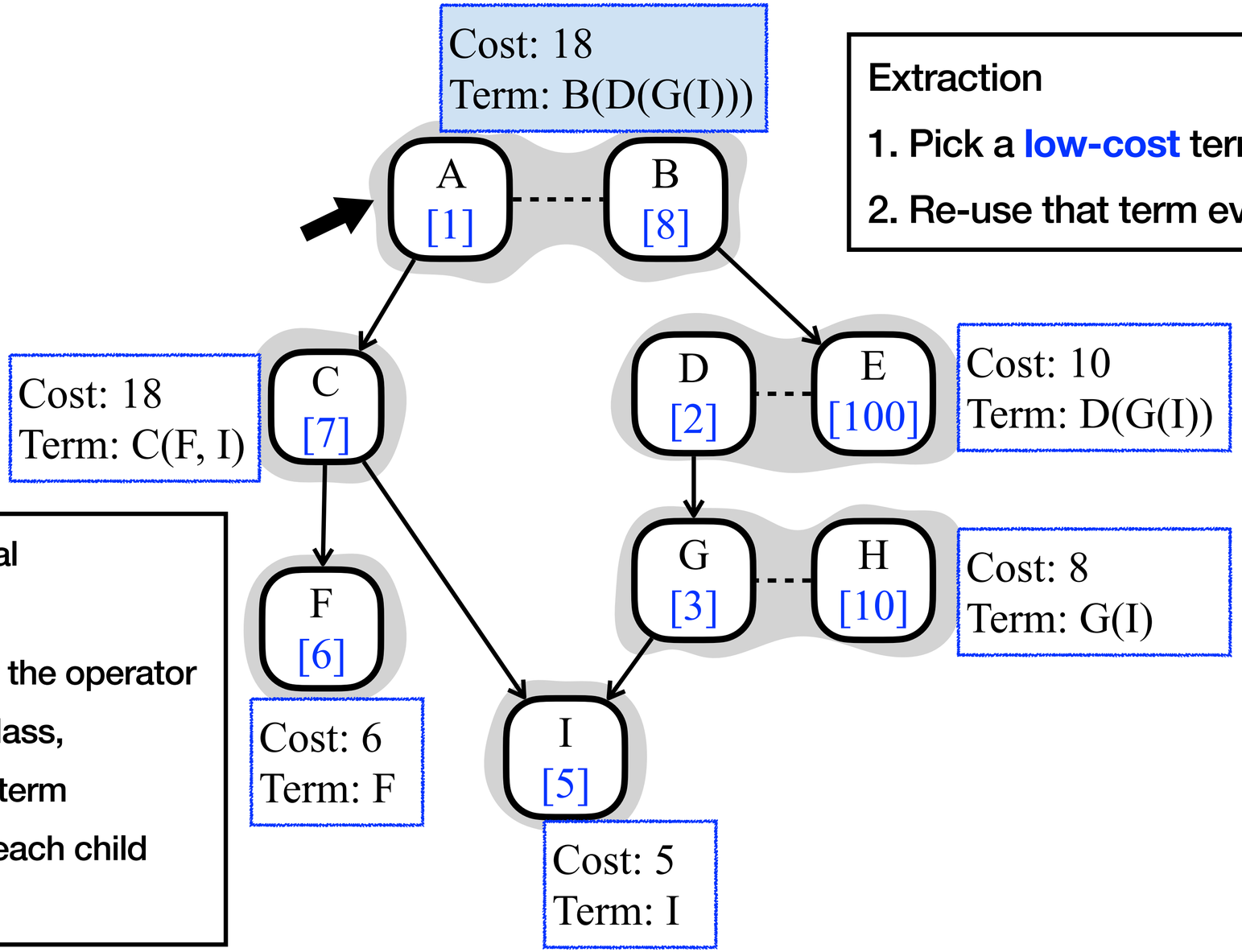
Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Bottom Up Graph Traversal

1. Initialize all costs to ∞
2. Leaf terms have cost of the operator
3. For each equivalence class,
   search for a lower cost term
   using the best cost for each child
4. Repeat until fixed point

Greedy Extraction with Cost Model



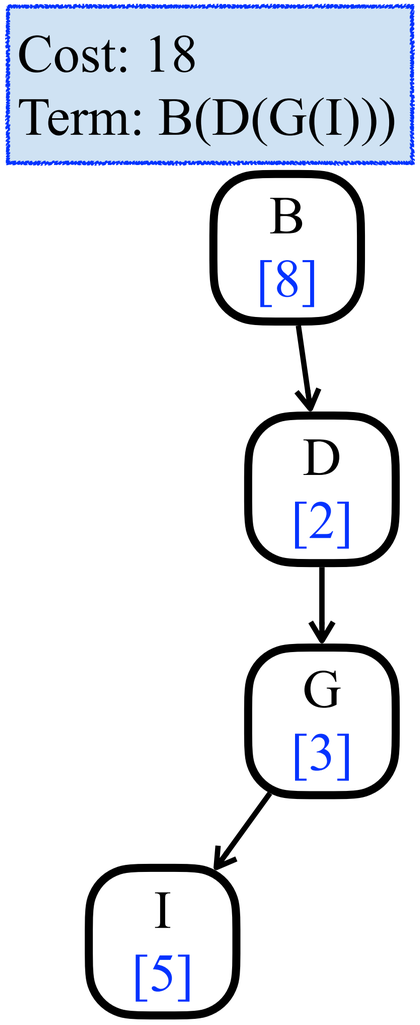
Extraction

1. Pick a **low-cost** term from each e-class
2. Re-use that term everywhere

Bottom Up Graph Traversal

1. Initialize all costs to ∞
2. Leaf terms have cost of the operator
3. For each equivalence class,
   search for a lower cost term
   using the best cost for each child
4. Repeat until fixed point

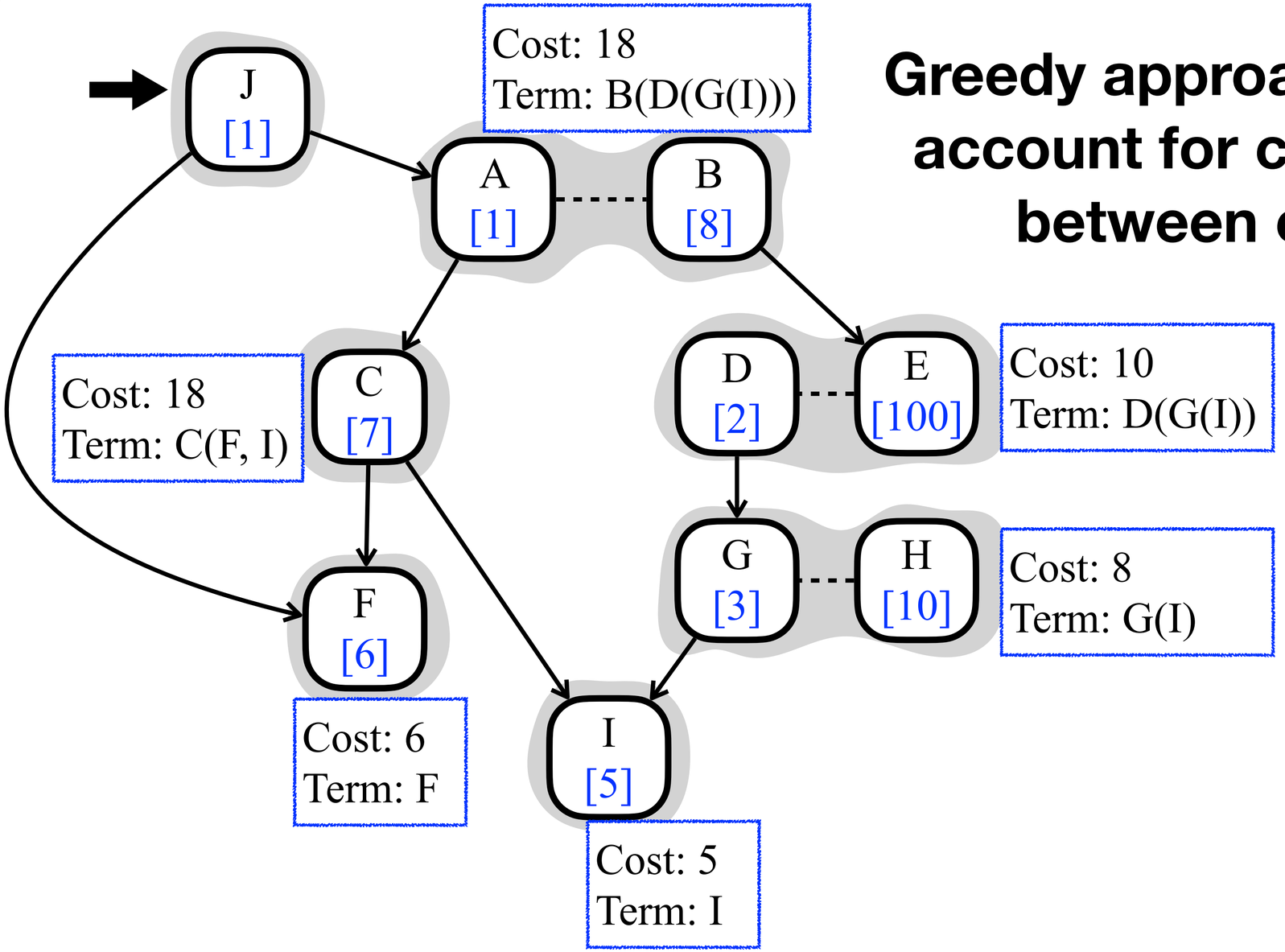
Greedy Extraction with Cost Model



Greedy Extraction with Cost Model

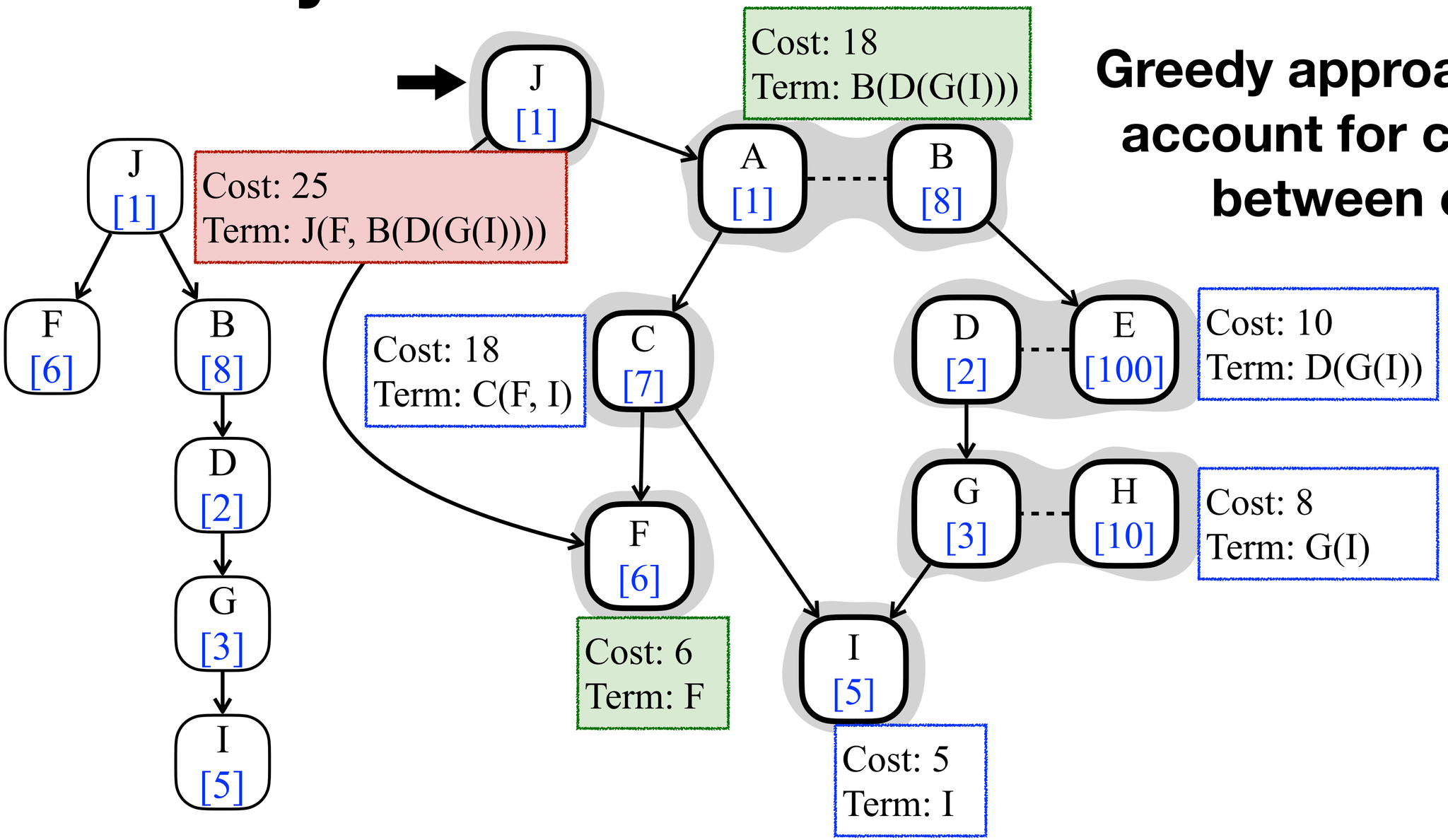
Greedy approach does not account for cost sharing between children

Greedy Extraction with Cost Model



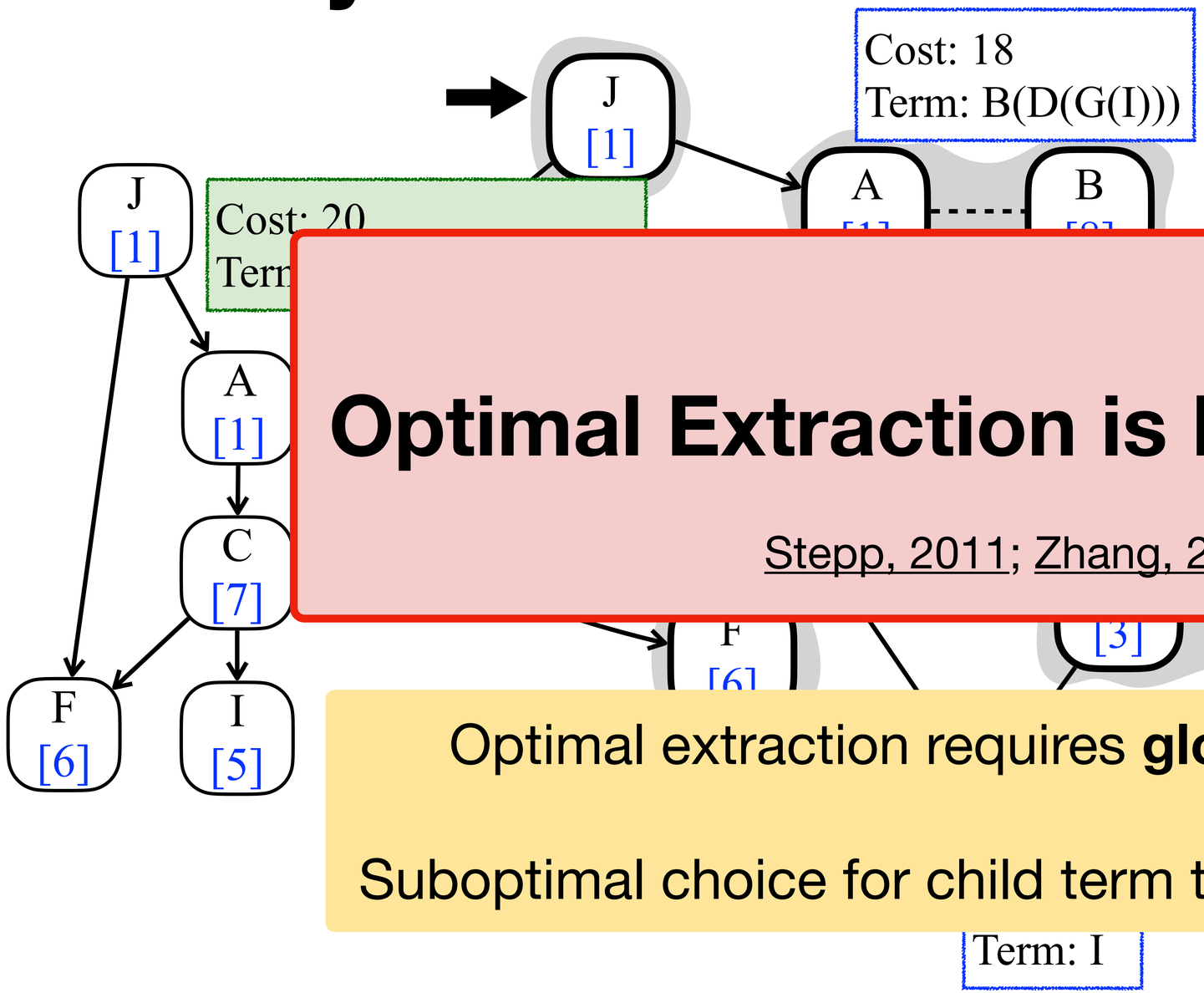
Greedy approach does not account for cost sharing between children

Greedy Extraction with Cost Model



Greedy approach does not account for cost sharing between children

Greedy Extraction with Cost Model



Greedy approach does not account for cost sharing between children

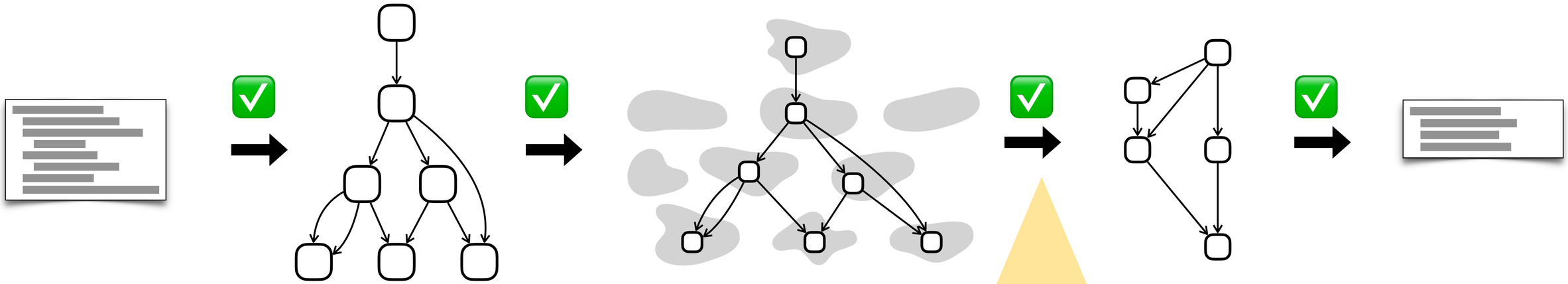
Optimal Extraction is NP Complete

Stepp, 2011; Zhang, 2023

Optimal extraction requires **global reasoning:**

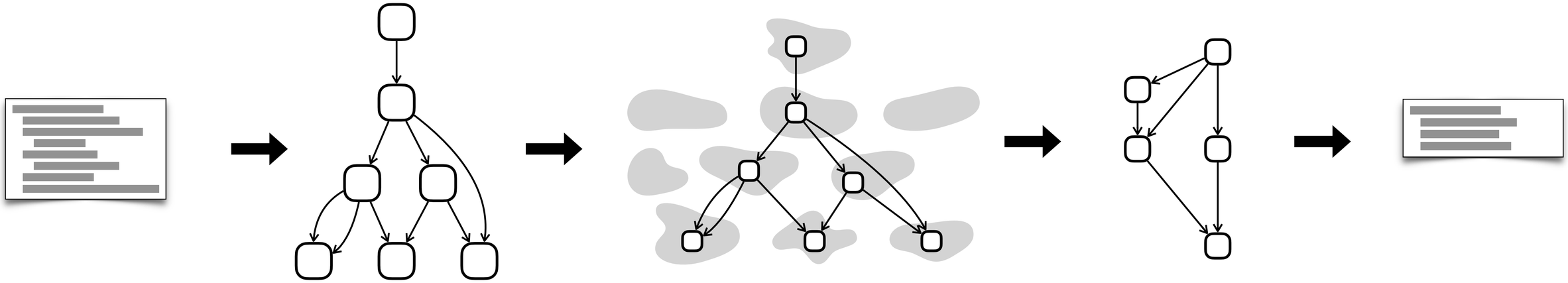
Suboptimal choice for child term to maximize sharing

Program Optimization with E-Graphs



Optimal: NP Complete
Greedy: Empirically Good

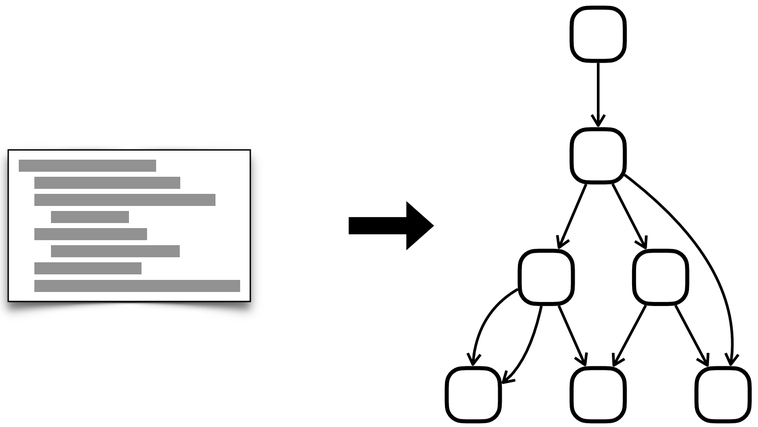
Program Optimization with E-Graphs

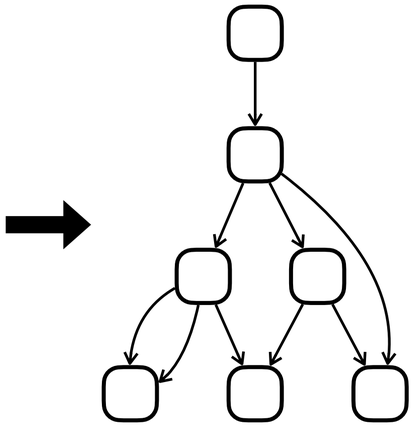


Can we use this approach for programs with side effects?

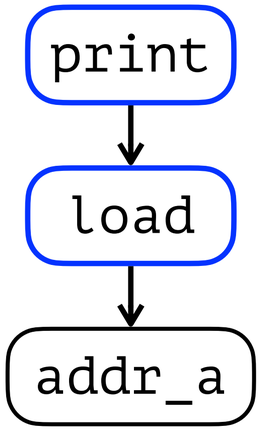
Memory operations, input/output, mutating state, and more

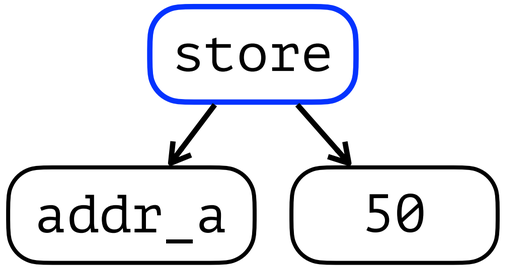
```
print(100)

store(addr_a, 50)

print(load(addr_a))
```

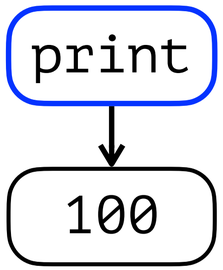


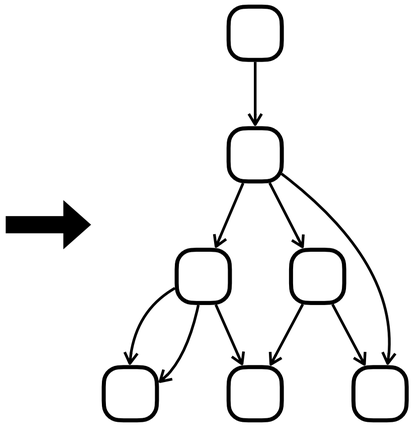


```
print(100)

store(addr_a, 50)

print(load(addr_a))
```

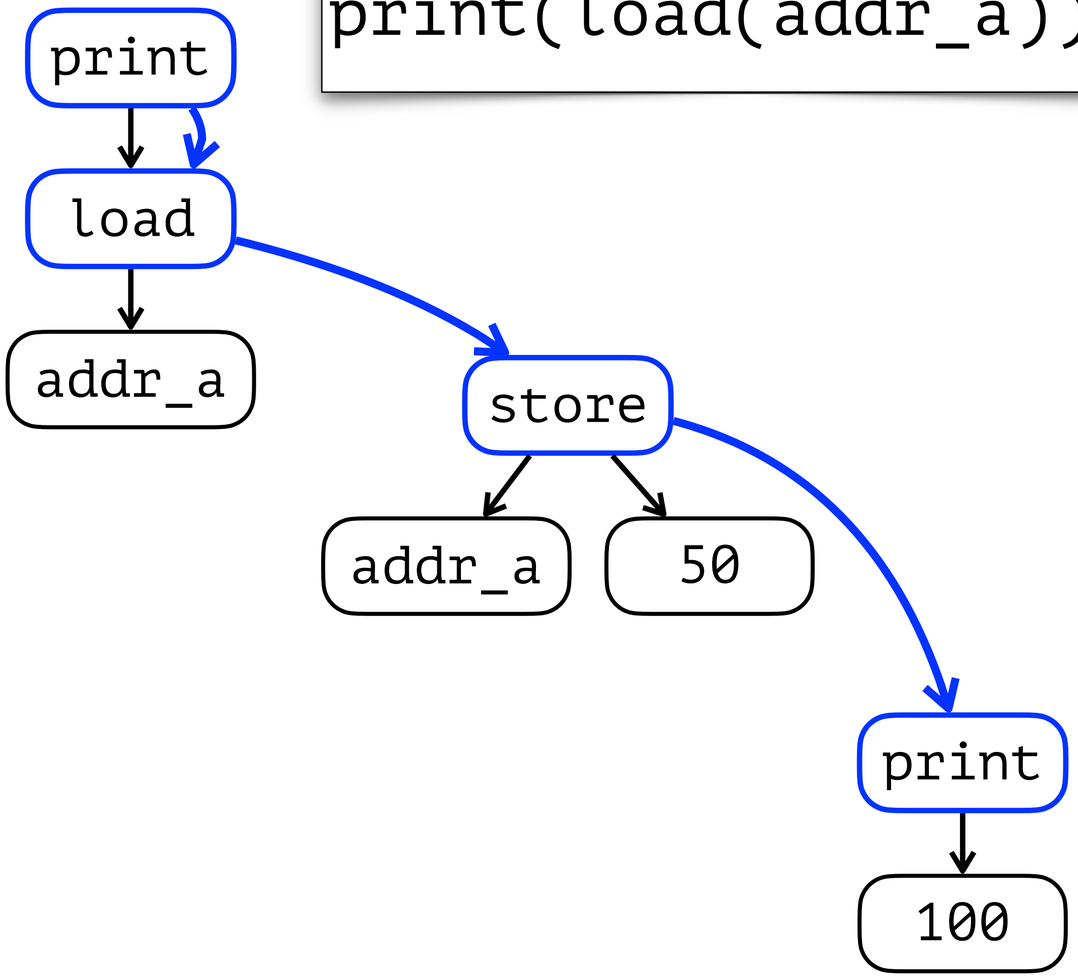




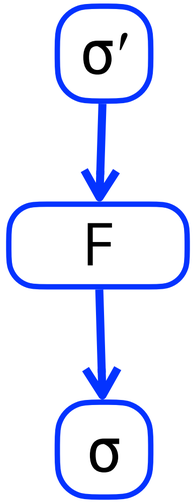




```
print(100)

store(addr_a, 50)

print(load(addr_a))
```

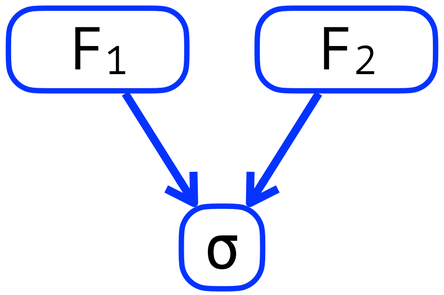


State Values



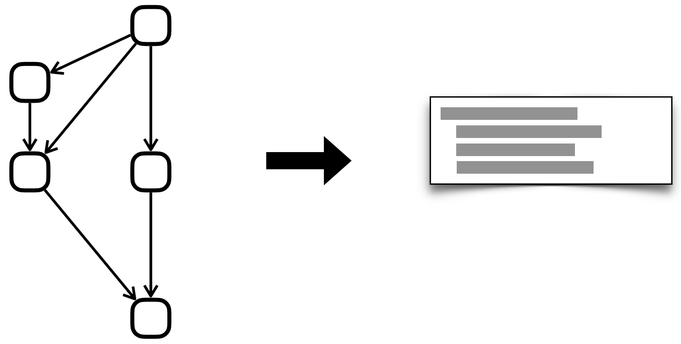
Any operator that takes a state value as input must produce a new state value as output

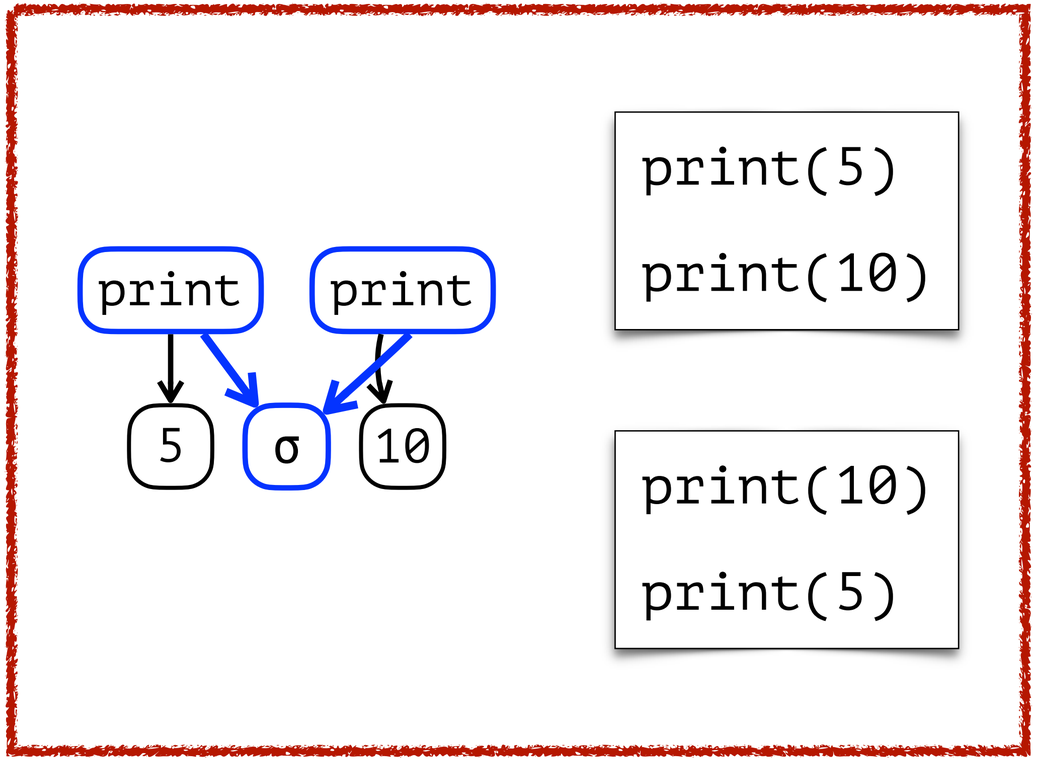
State Values



“Forking” the state value is not allowed

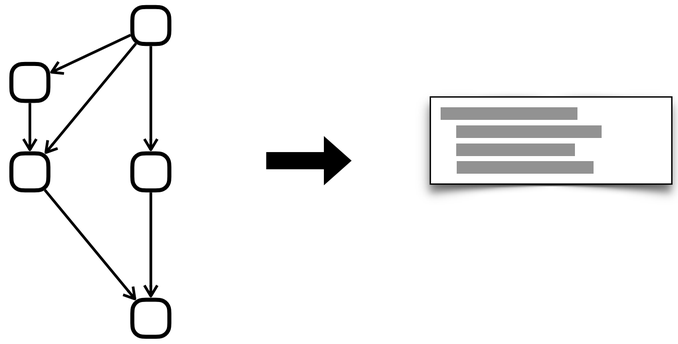
State Values

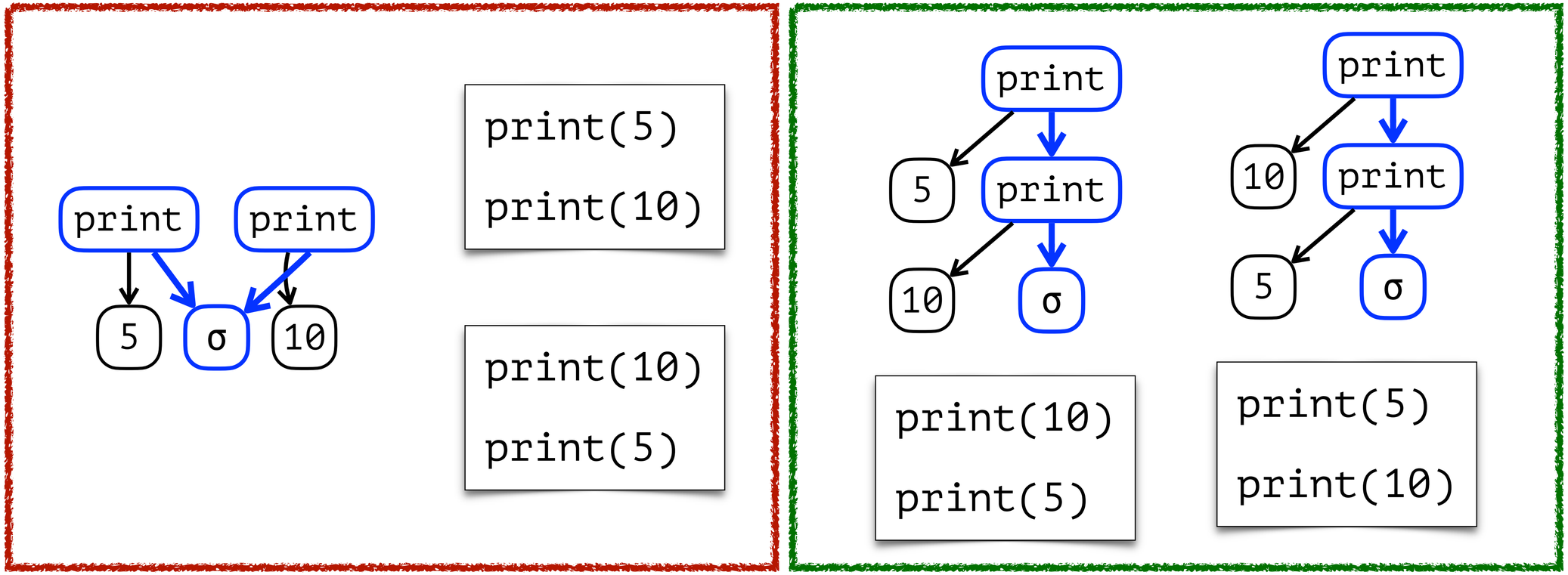




“Forking” the state value is not allowed

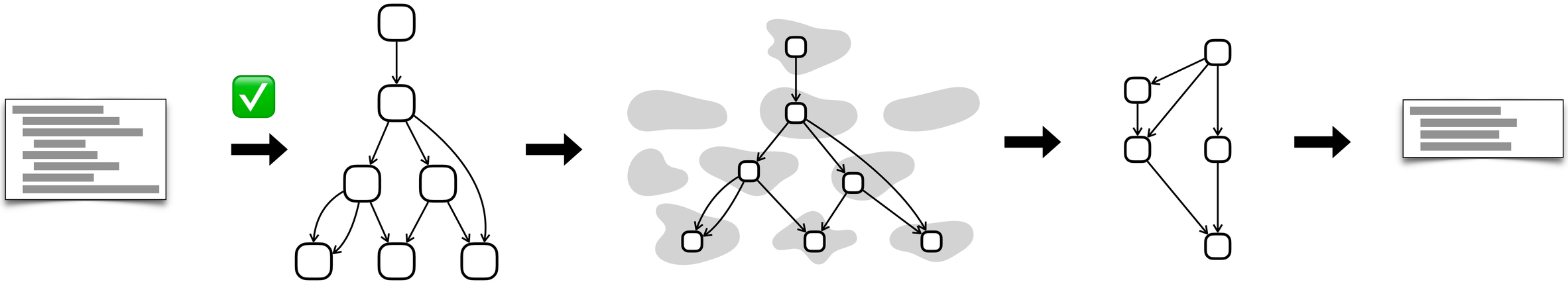
State Values



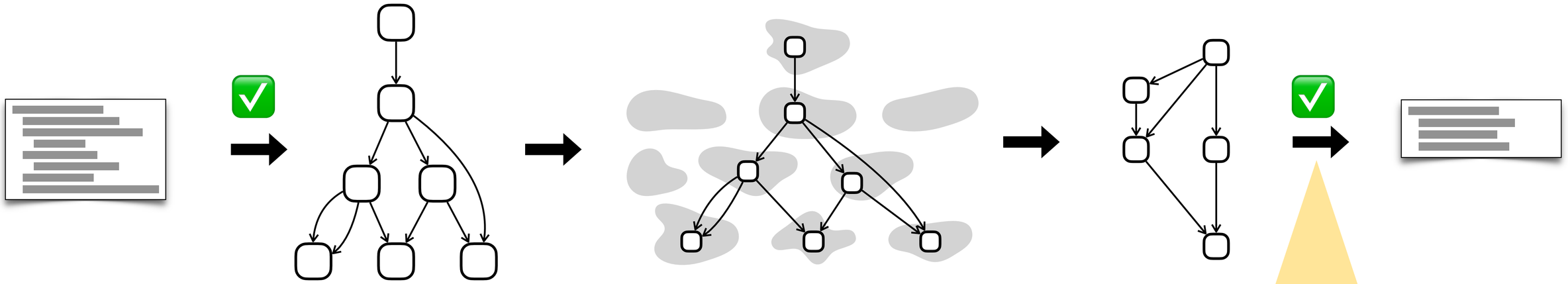


“Forking” the state value is not allowed

Effectful Program Optimization with E-Graphs



Effectful Program Optimization with E-Graphs



Reconstruction succeeds if effectful operations form a linear sequence

Program Equivalence with Side Effects

```
def foo(x):
  print(x)
  return x * 2

a = foo(10)
b = foo(5) + foo(5)
```

Are a and b equivalent?

Yes! a and b are both 20

No! They print different things

Two Equivalence Relations

Terms are equivalent if they have the same value.

Important for optimization

Terms are equivalent if they have the same value **and the same side effects.**

Important for correctness

Update

```
x = update(a, v)
```

Update

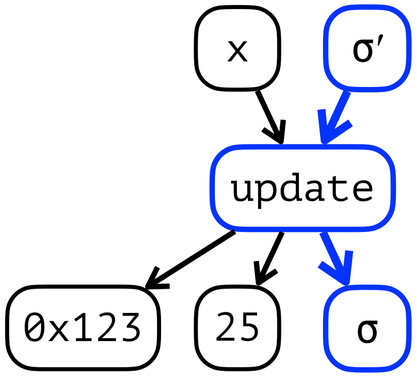
```
x = update(a, v)
```

Writes `v` to memory location `a`,
returns the previous value stored there

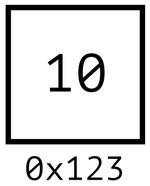
Update

```
x = update(a, v)
```

Writes v to memory location a,
returns the previous value stored there



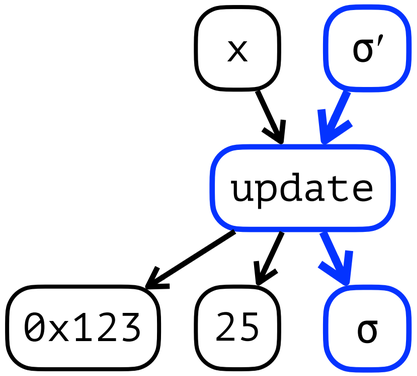

```
x = update(0x123, 25)
```



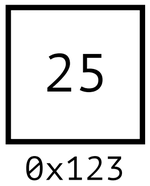
Update

```
x = update(a, v)
```

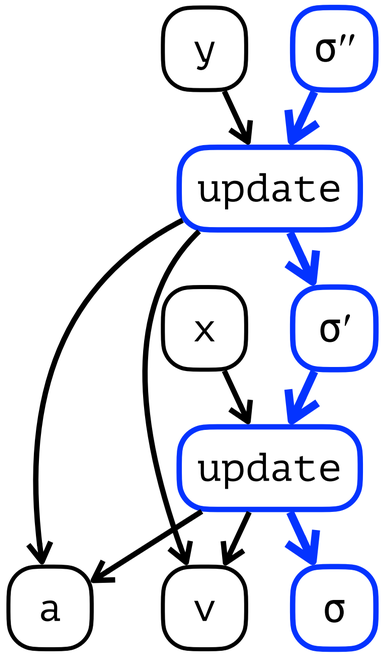
Writes v to memory location a,
returns the previous value stored there




```
x = update(0x123, 25)

// x is 10
```



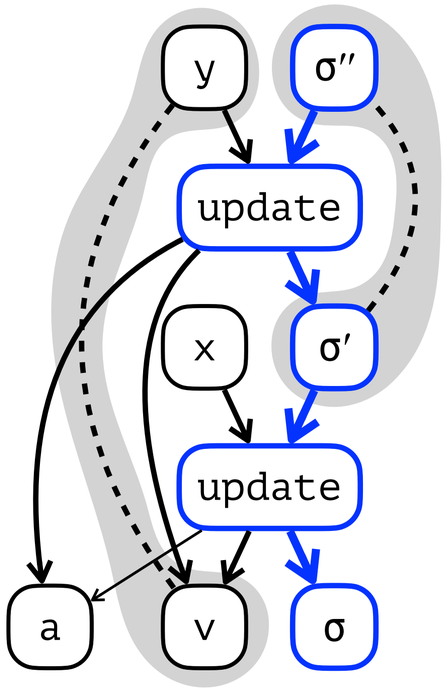
Redundant Update Elimination




```
x = update(a, v)

y = update(a, v)
```

update is idempotent

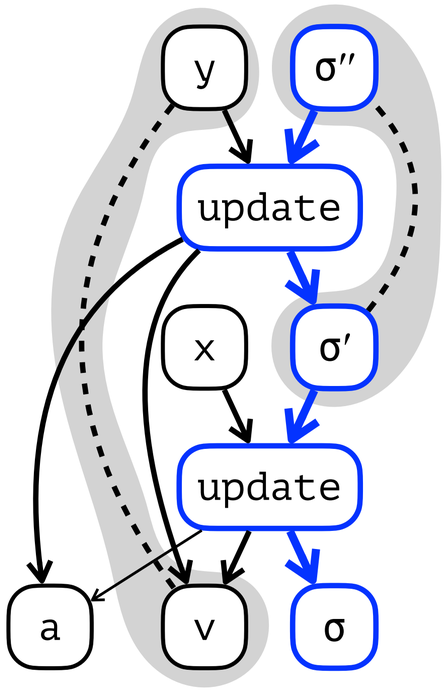
Redundant Update Elimination




```
x = update(a, v)
y = update(a, v)
```

update is idempotent

Redundant Update Elimination

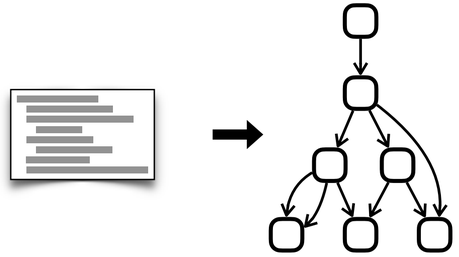



```
x = update(a, v)
y = update(a, v)
```

update is idempotent

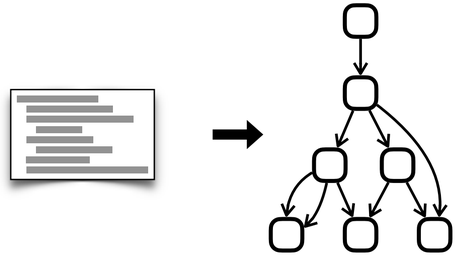
Is this optimization safe?

Redundant Update Elimination

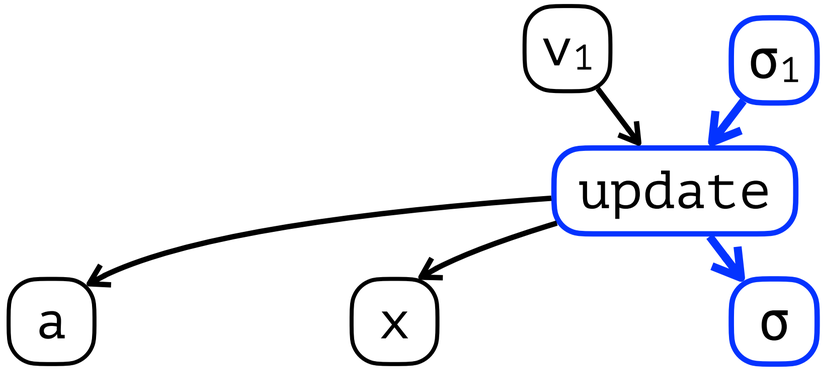


```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

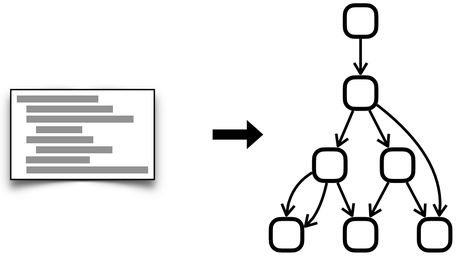
Redundant Update Elimination



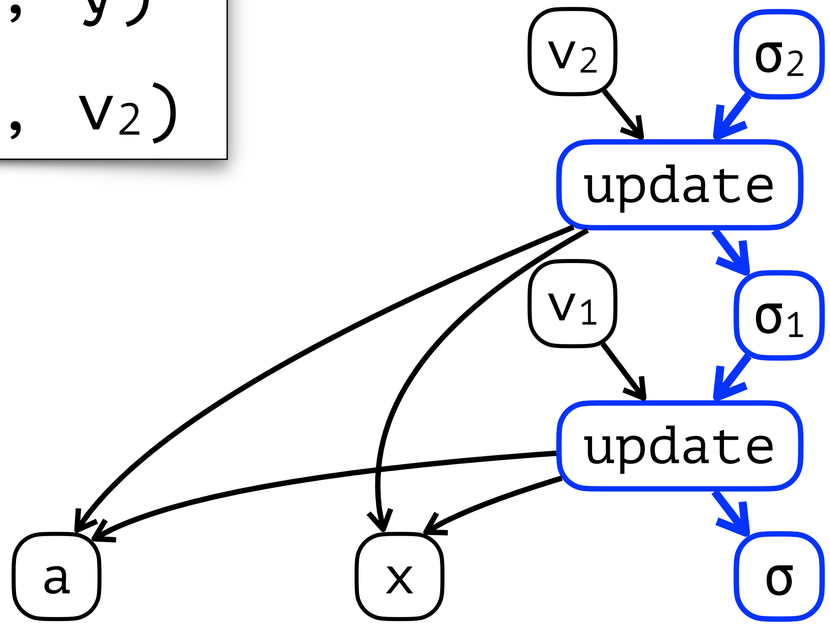
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



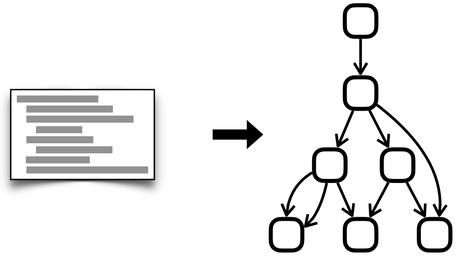
Redundant Update Elimination



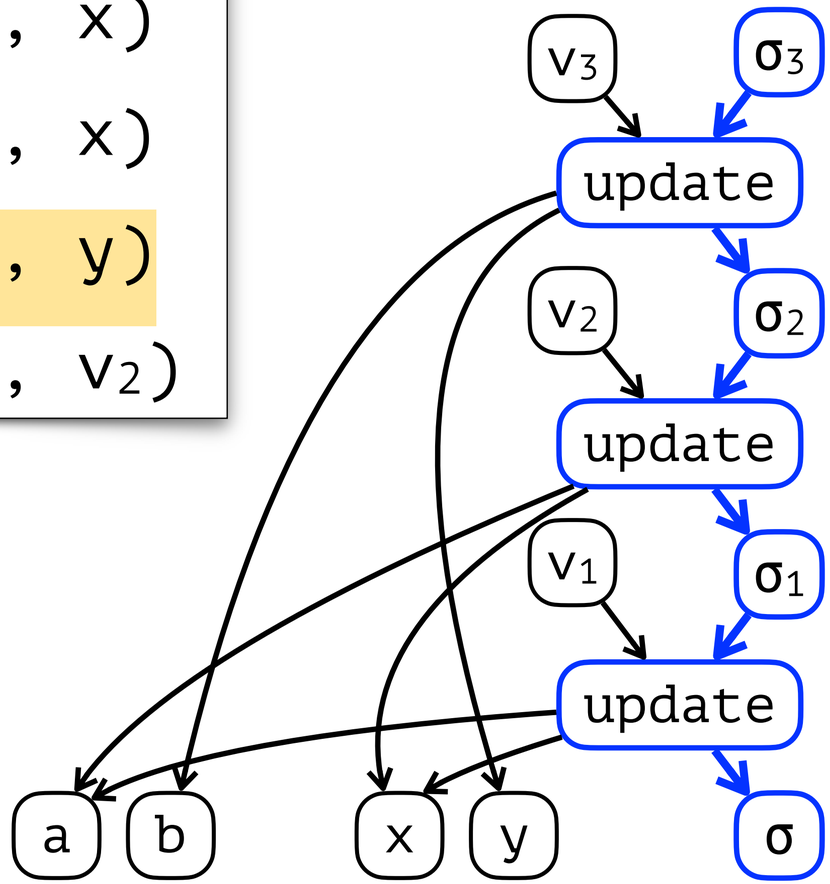
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



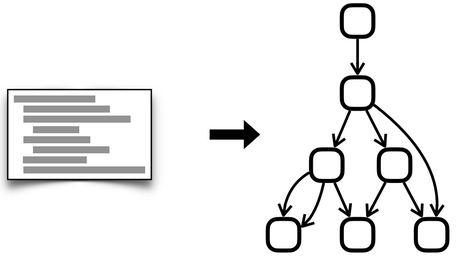
Redundant Update Elimination



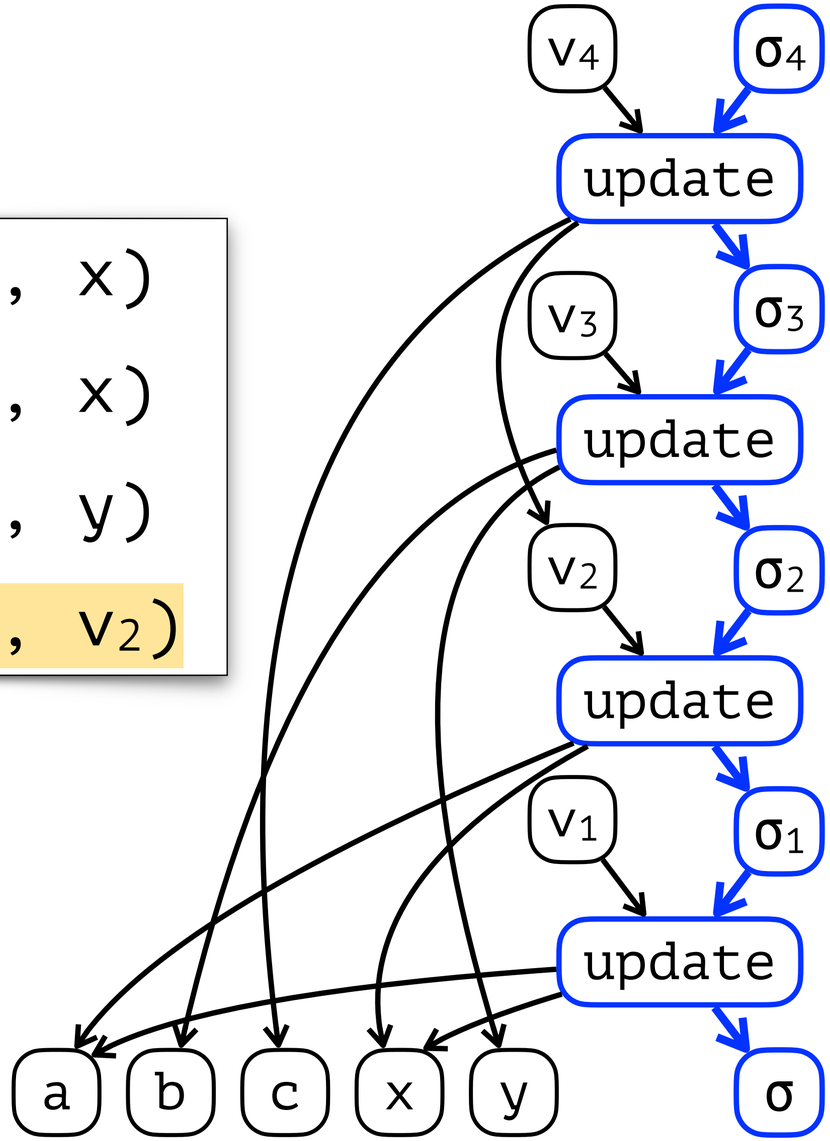
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



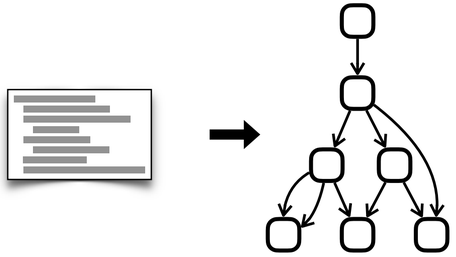
Redundant Update Elimination



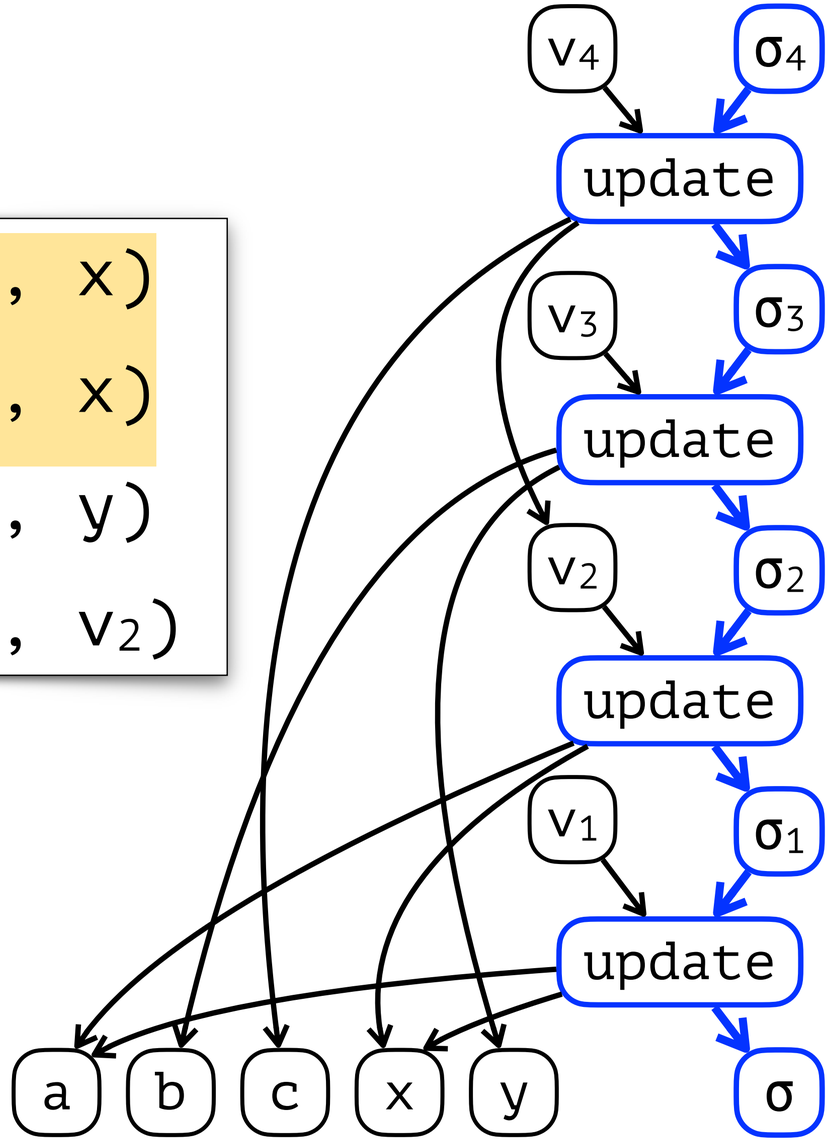
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



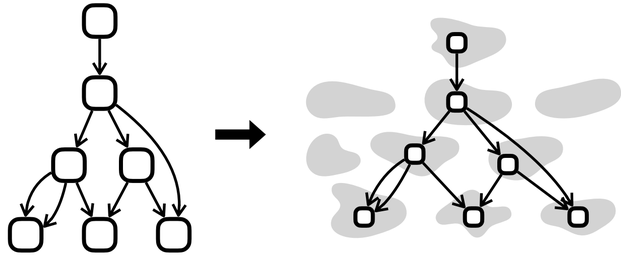
Redundant Update Elimination



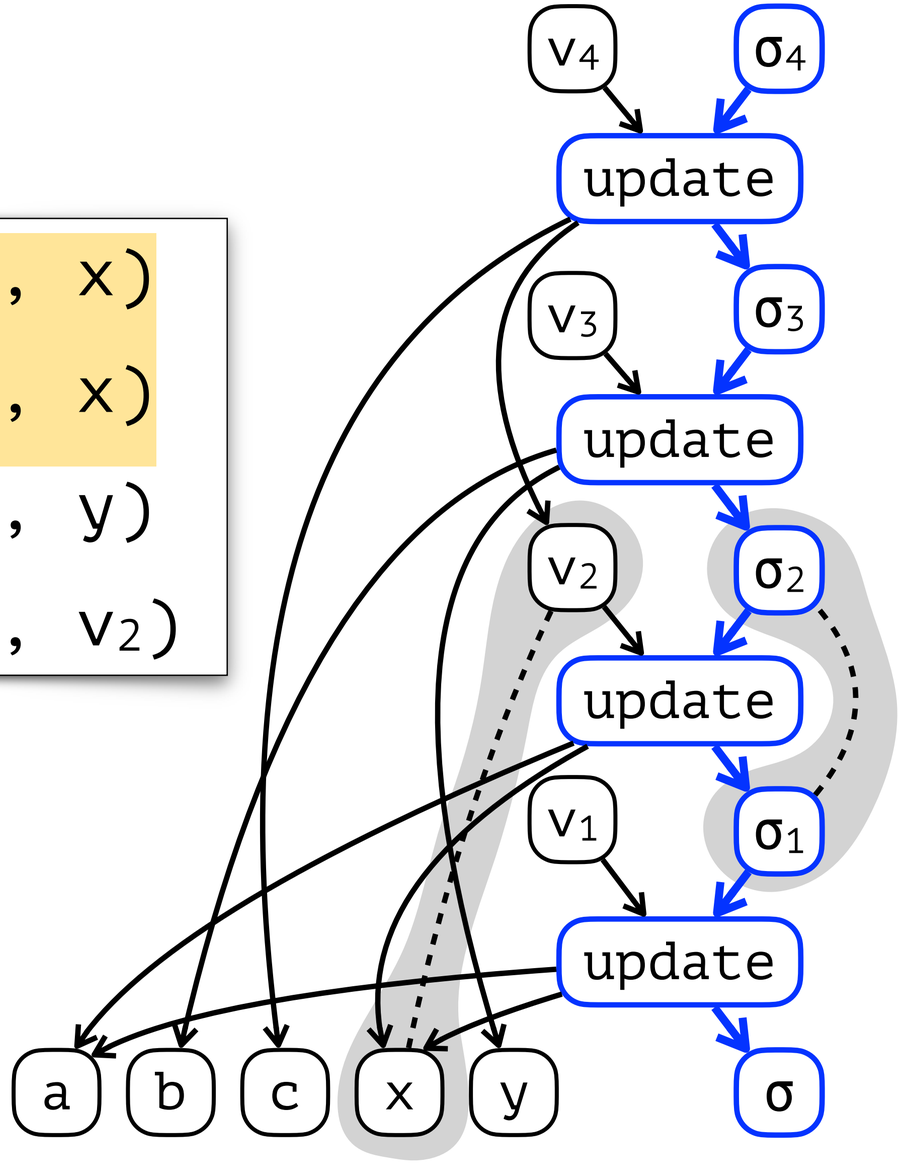
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



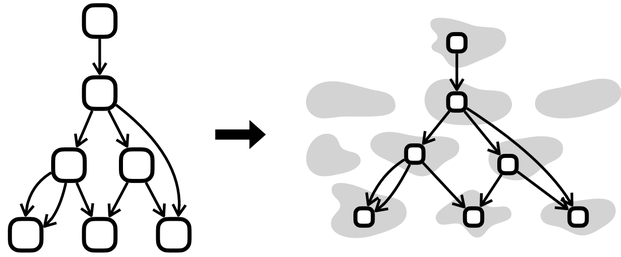
Redundant Update Elimination



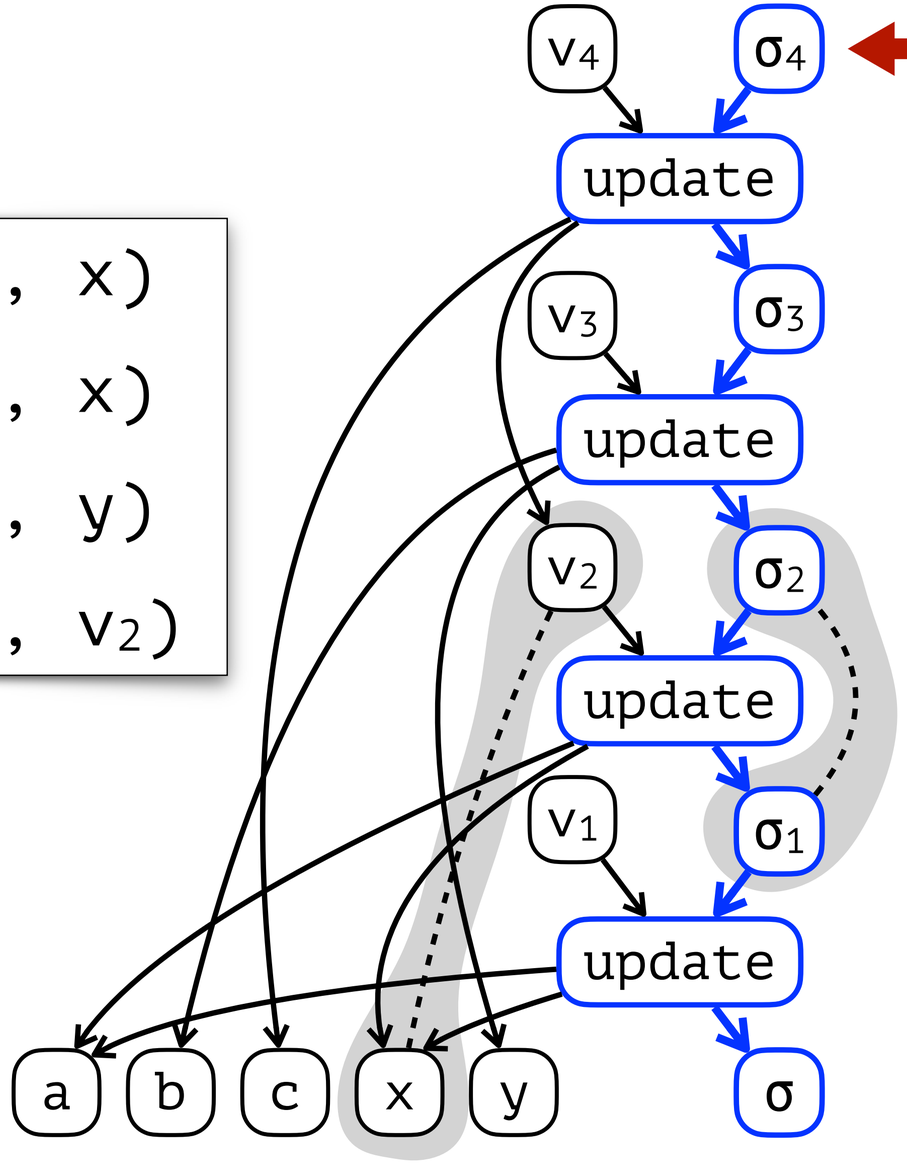
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



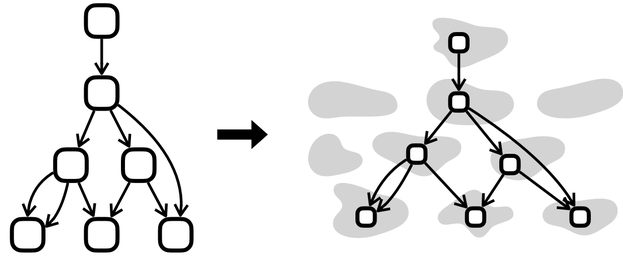
Redundant Update Elimination



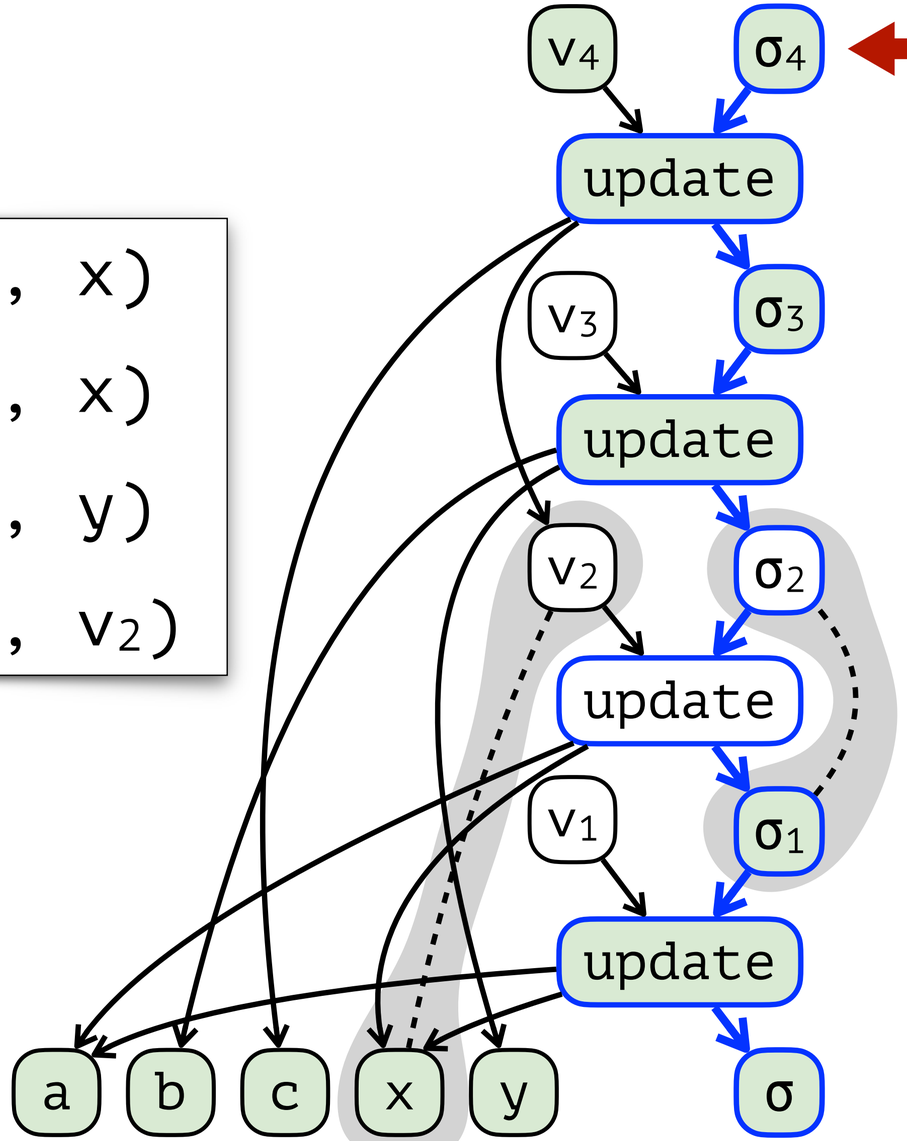
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



Redundant Update Elimination

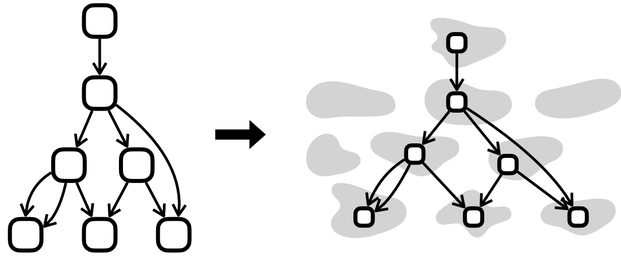


```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

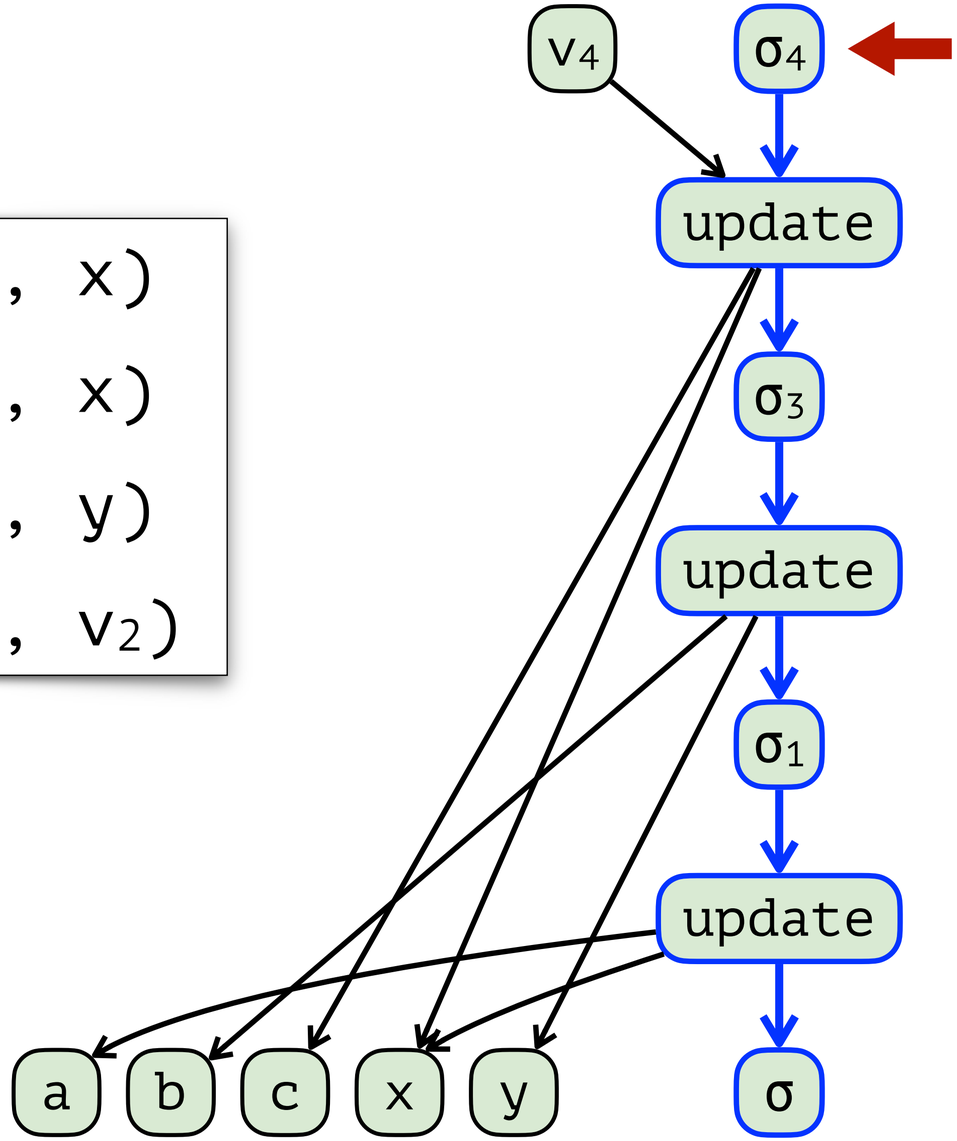


Extraction Option 1

Redundant Update Elimination

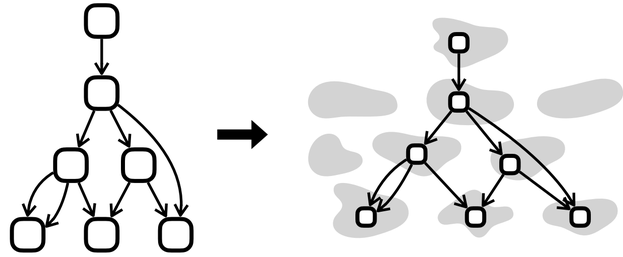


```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

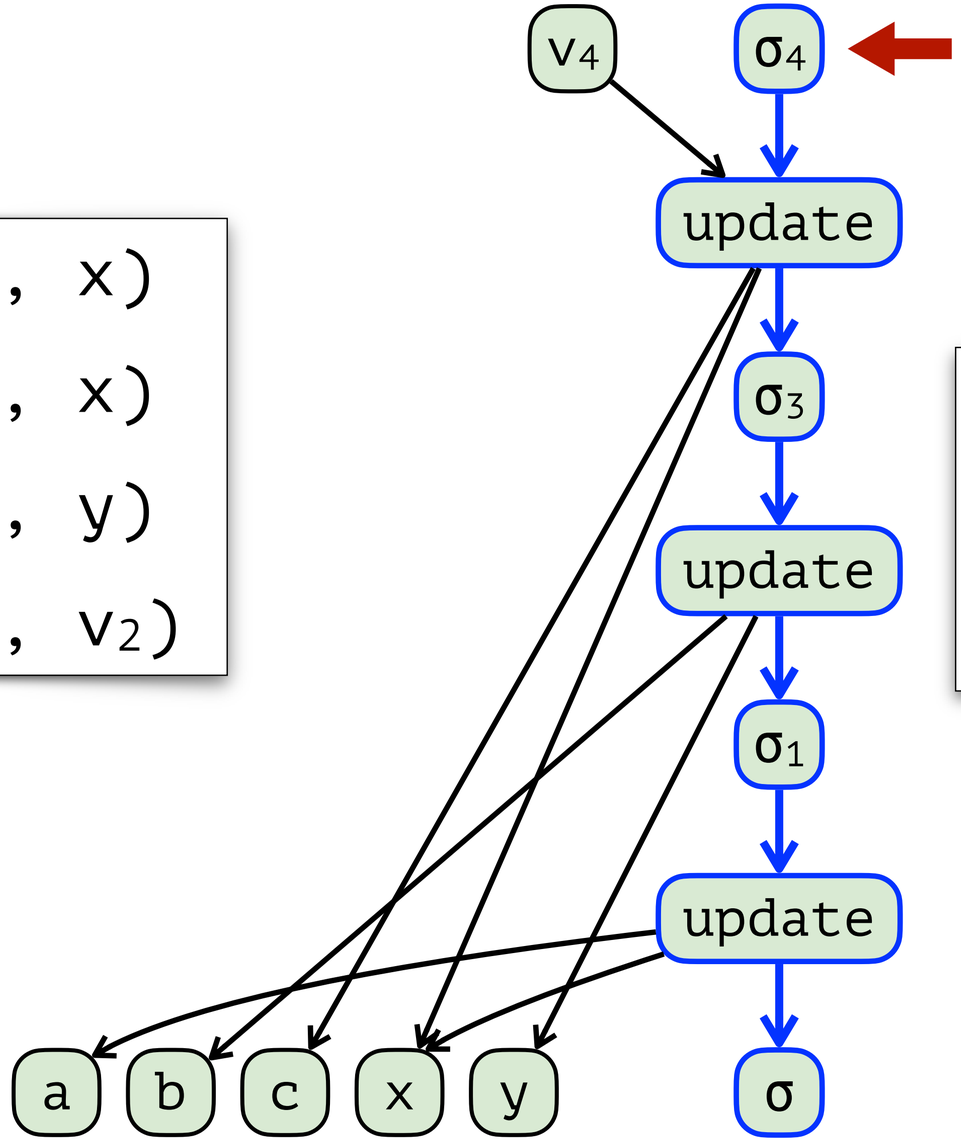


Extraction Option 1

Redundant Update Elimination



```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

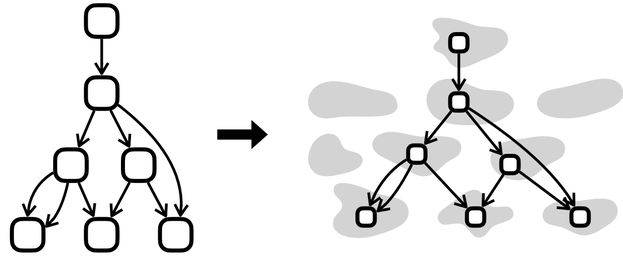



```
(v1, σ1) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, x)
```

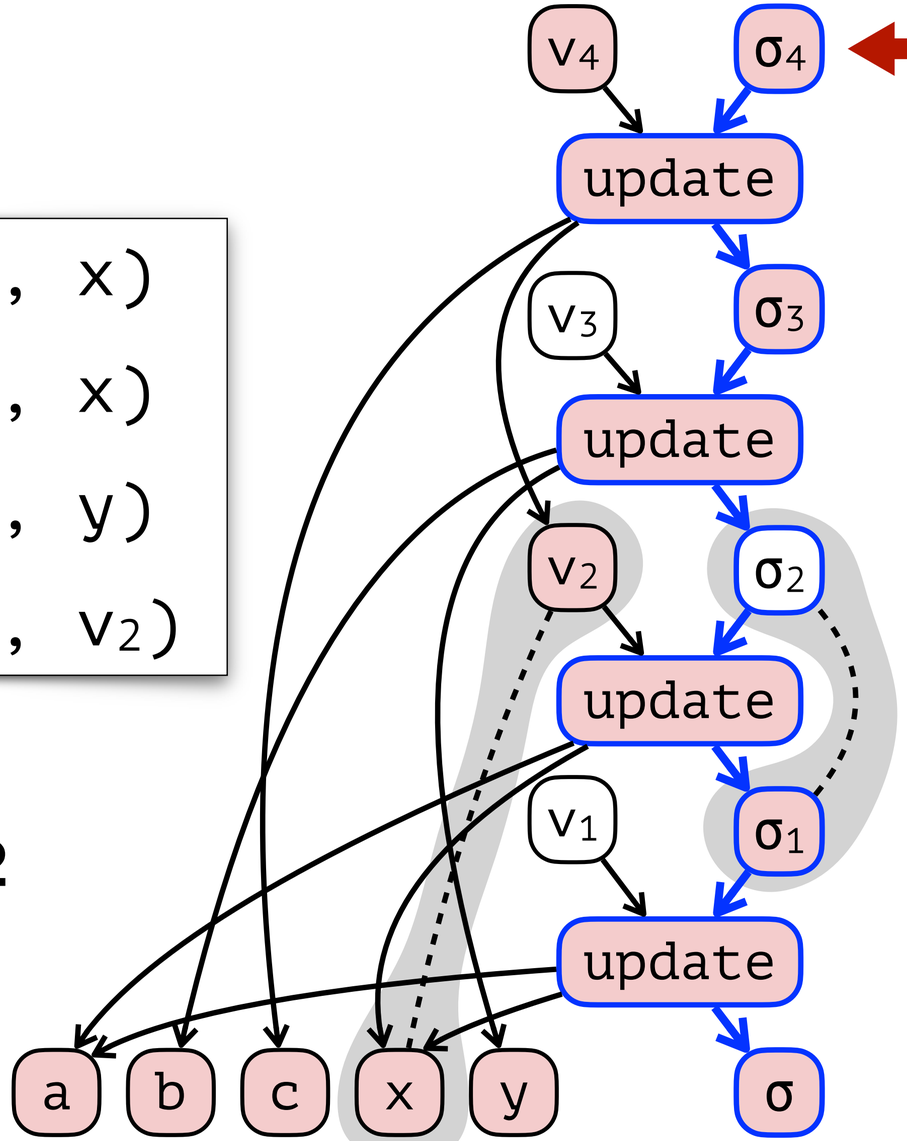
✅

Extraction Option 1

Redundant Update Elimination

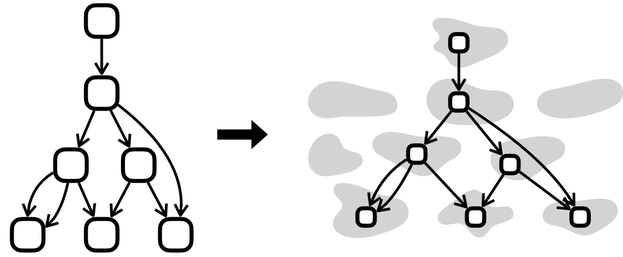


```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

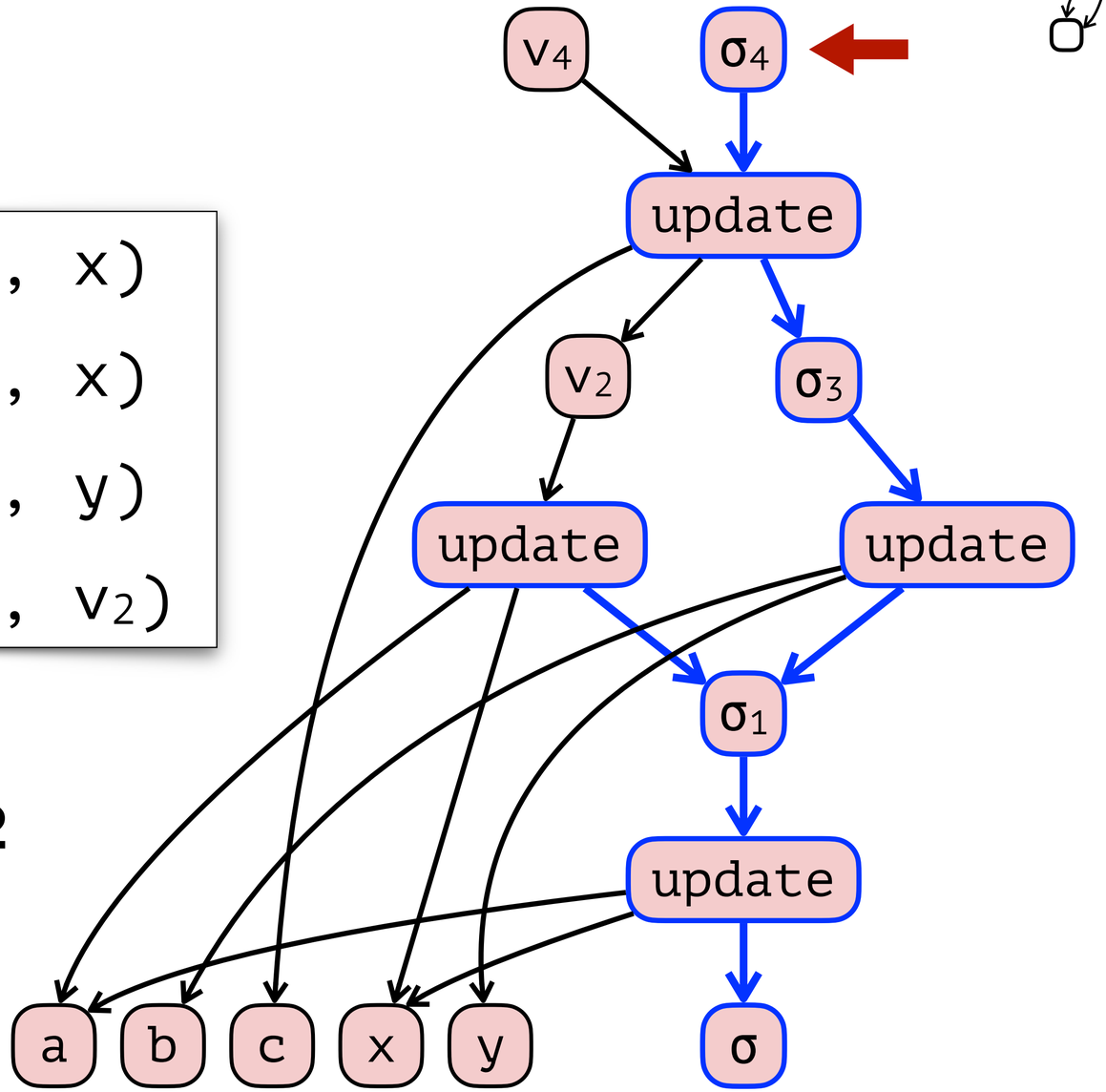


Extraction Option 2

Redundant Update Elimination

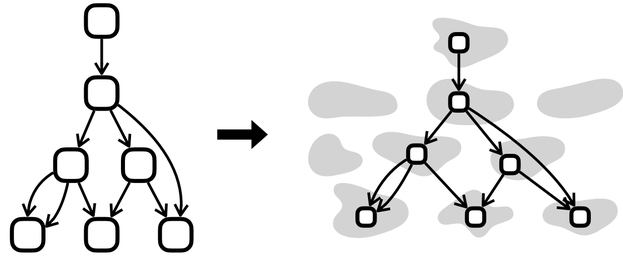


```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

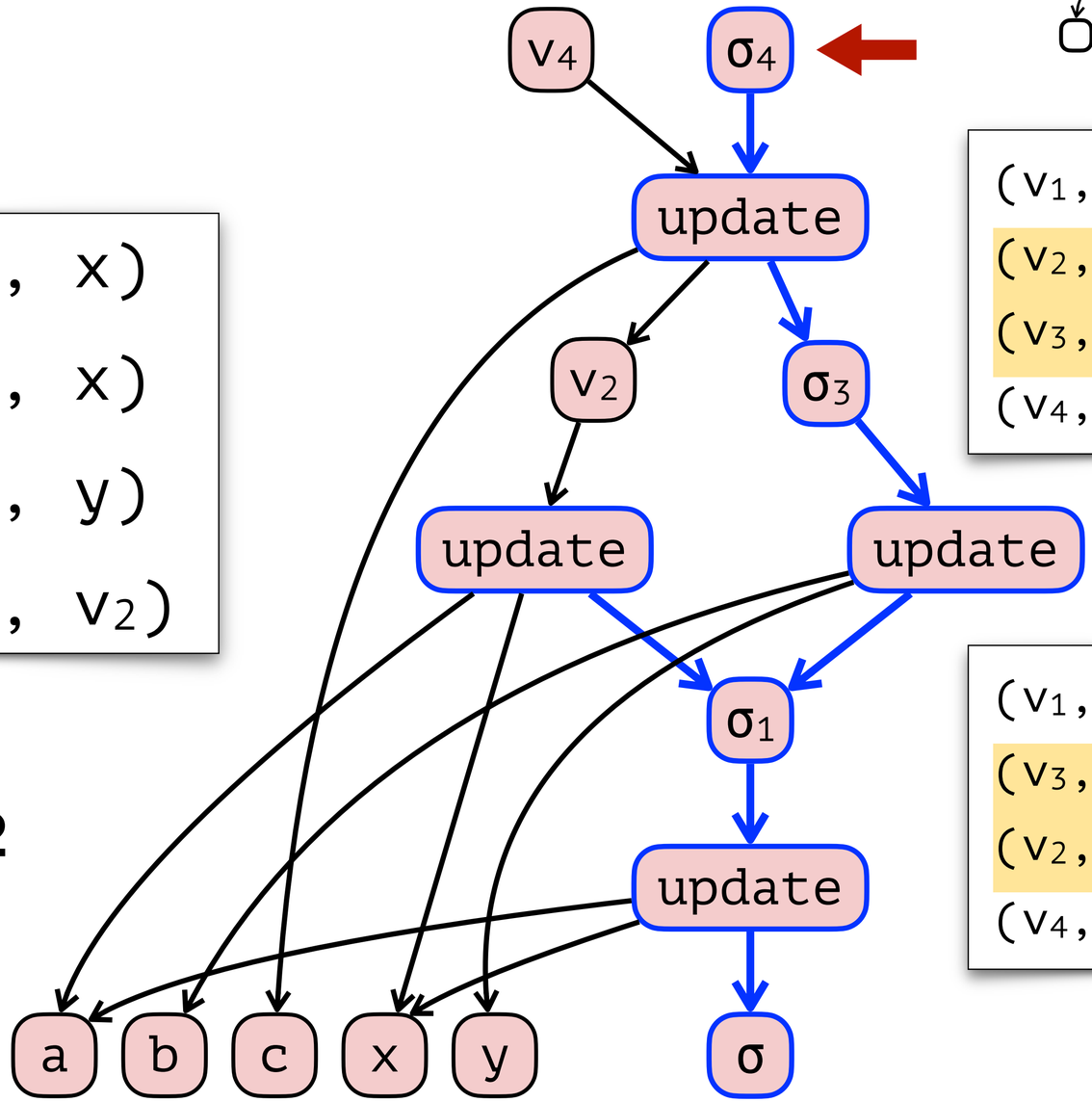


Extraction Option 2

Redundant Update Elimination



```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```



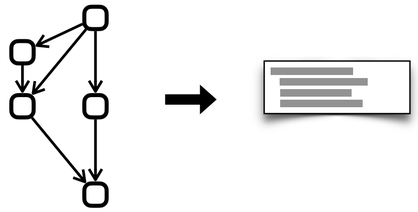

```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

Which program?

```
(v1, σ1) = update(a, x)
(v3, σ3) = update(b, y)
(v2, σ2) = update(a, x)
(v4, σ4) = update(c, v2)
```

Extraction Option 2

Redundant Update Elimination



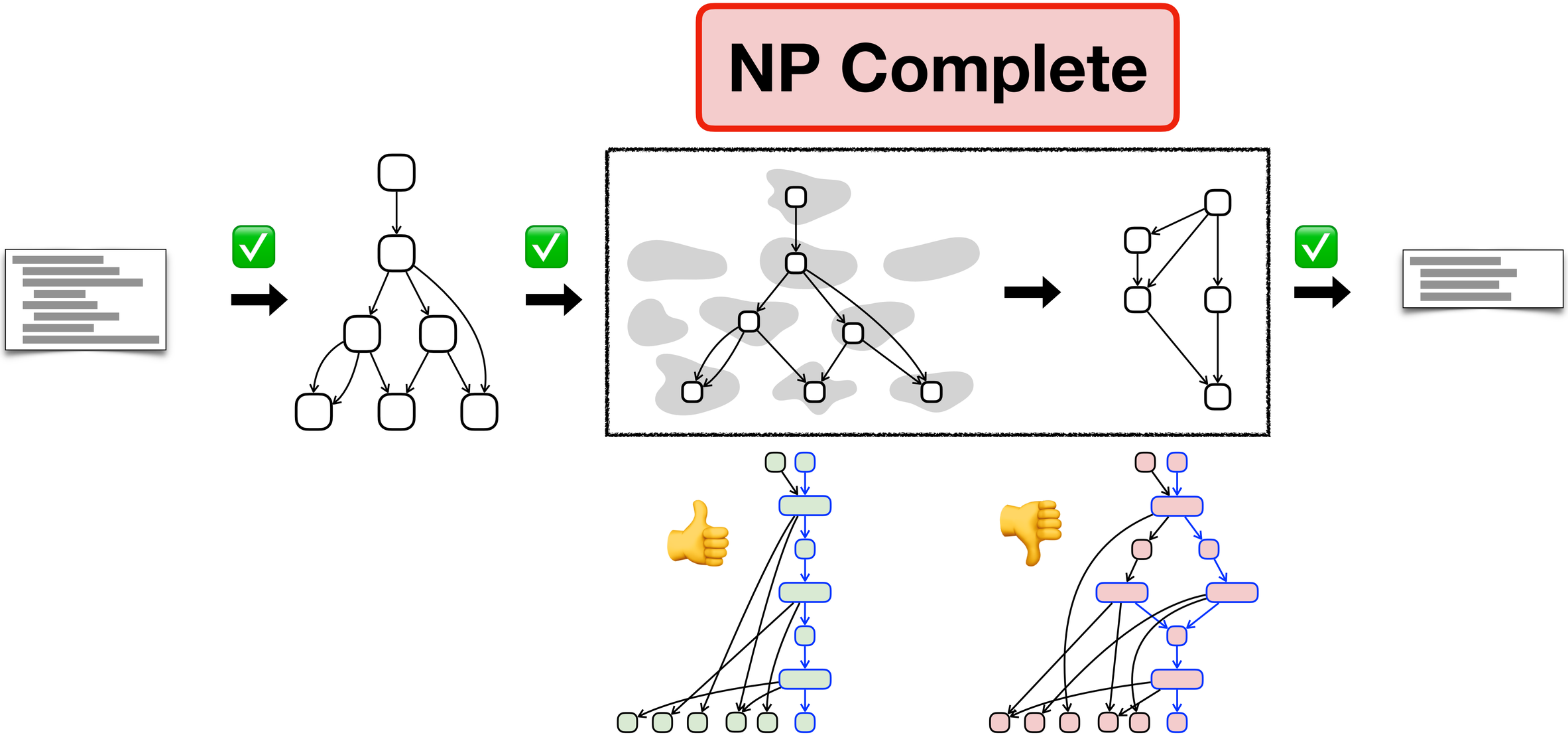
```
(v1, σ1) = update(a, x)
(v2, σ2) = update(a, x)
(v3, σ3) = update(b, y)
(v4, σ4) = update(c, v2)
```

```
(v1, σ1) = update(a, x)
(v3, σ3) = update(b, y)
(v2, σ2) = update(a, x)
(v4, σ4) = update(c, v2)
```

Does it matter?

What if a and b are the same address?

Effect-safe Extraction



If Problem $\mathcal{A}$ is NP-Hard, then Problem $\mathcal{B}$ is NP-Hard

Effect-safe Extraction is NP-Hard

If Problem $\mathcal{A}$ is NP-Hard, then Effect-Safe Extraction is NP-Hard

Effect-safe Extraction is NP-Hard

If SAT is NP-Hard, then Effect-Safe Extraction is NP-Hard

Effect-safe Extraction is NP-Hard

Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

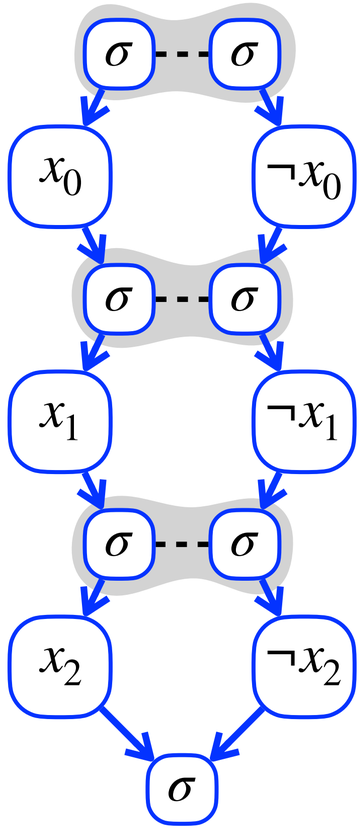
$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$



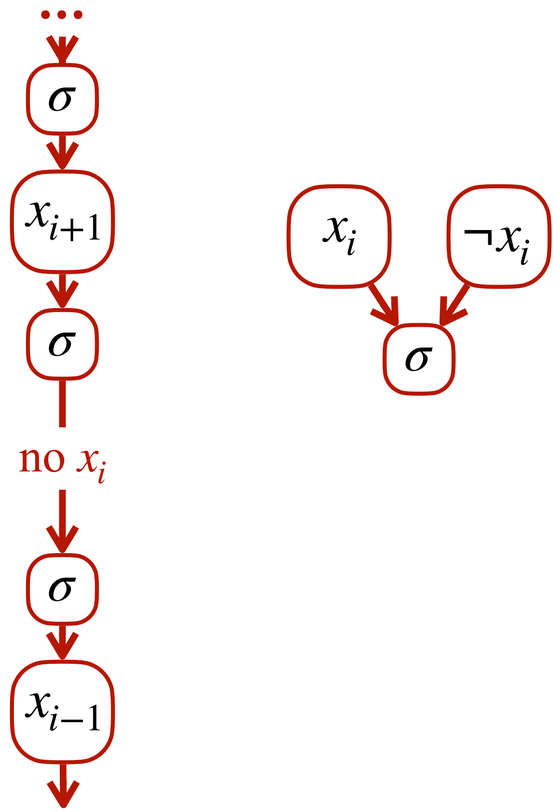
Proof Strategy

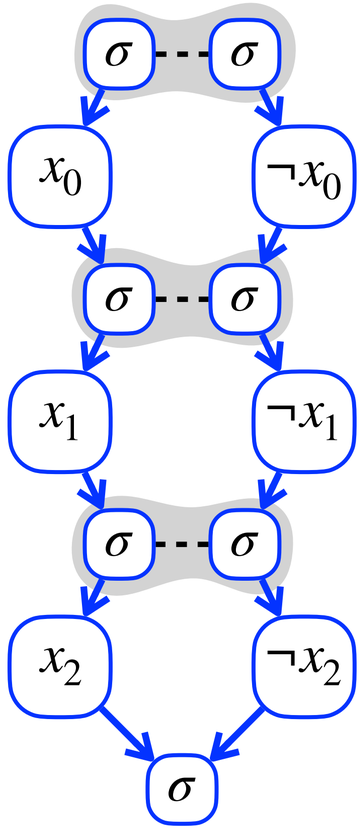
1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

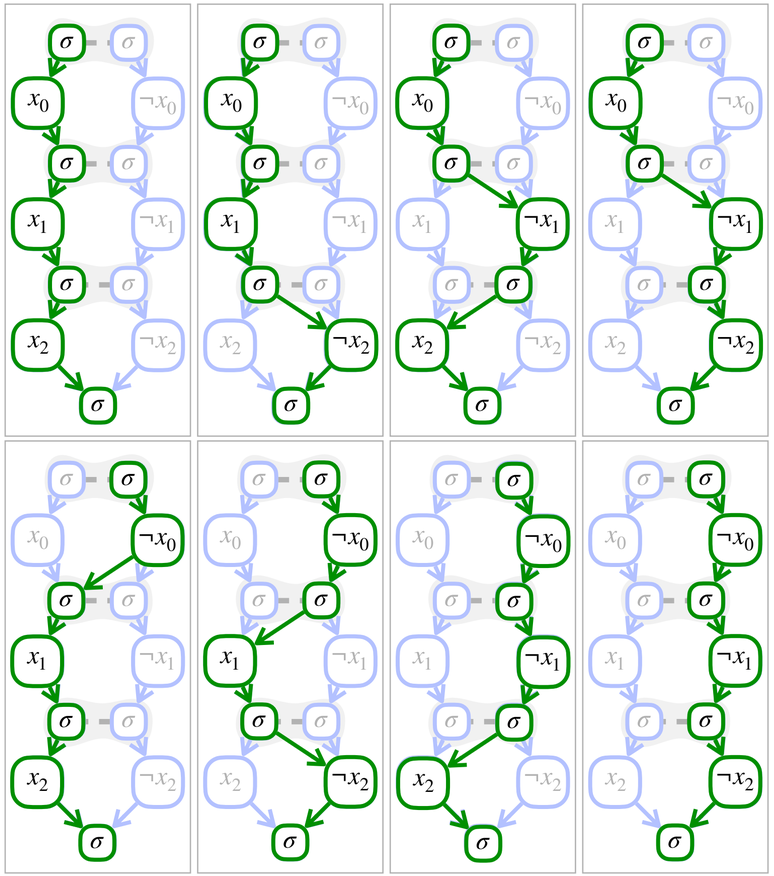
$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

Invalid Assignment ⇒ Not Effect-Safe



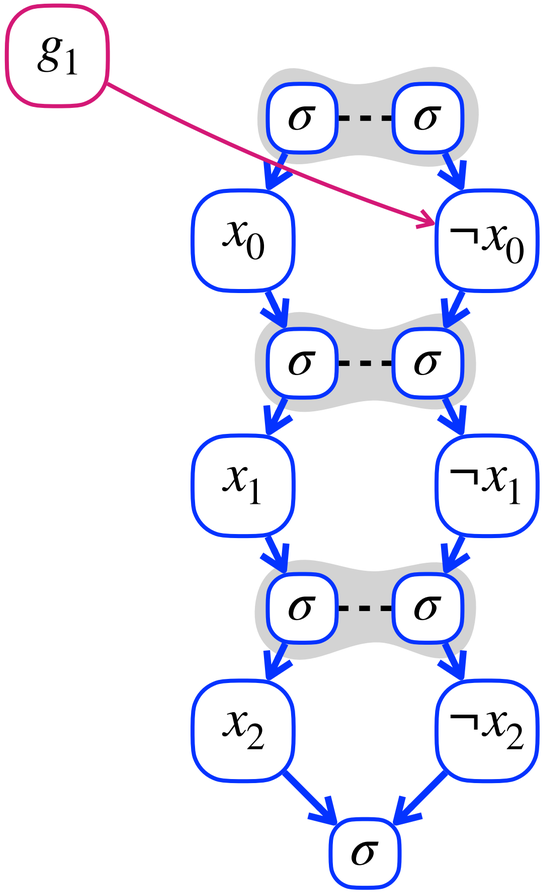


Valid Assignment ⇒ Effect-Safe



Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

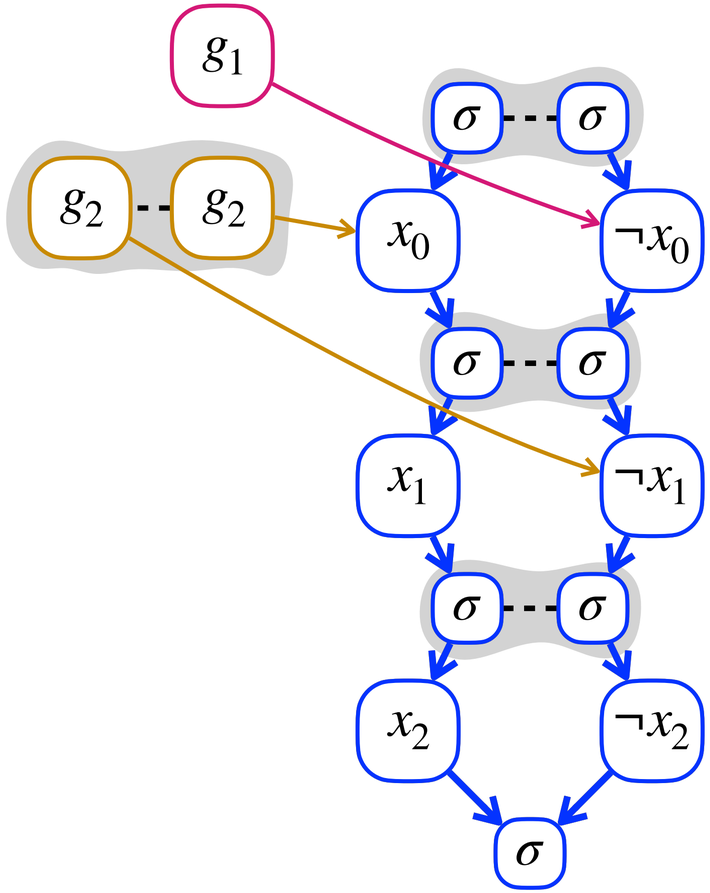


Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

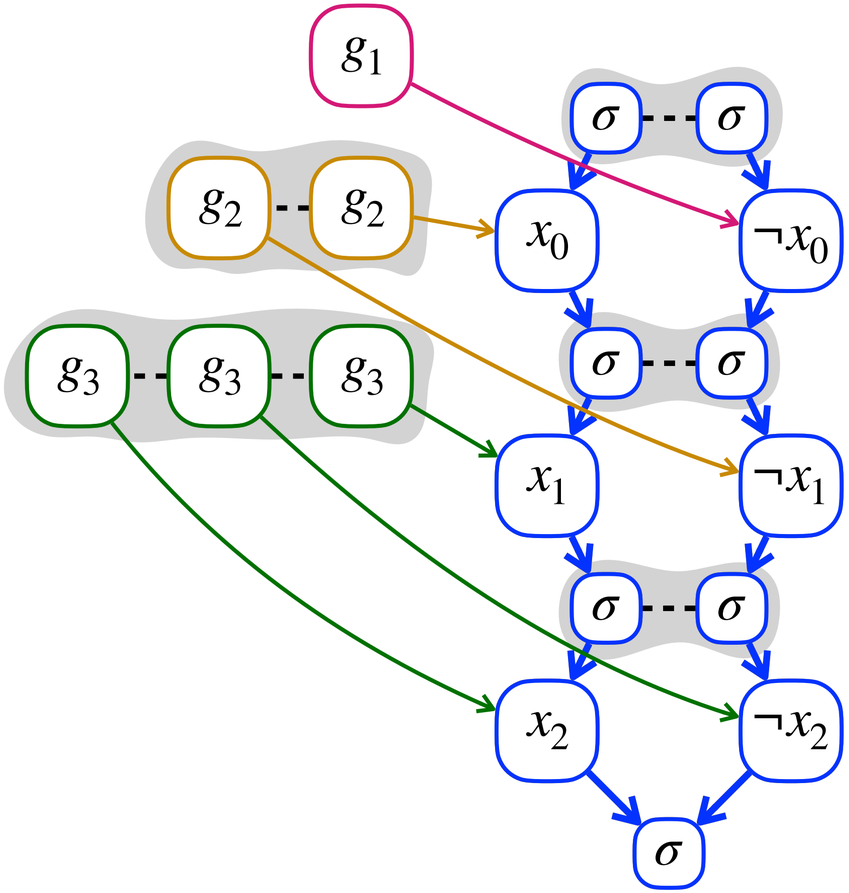


Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

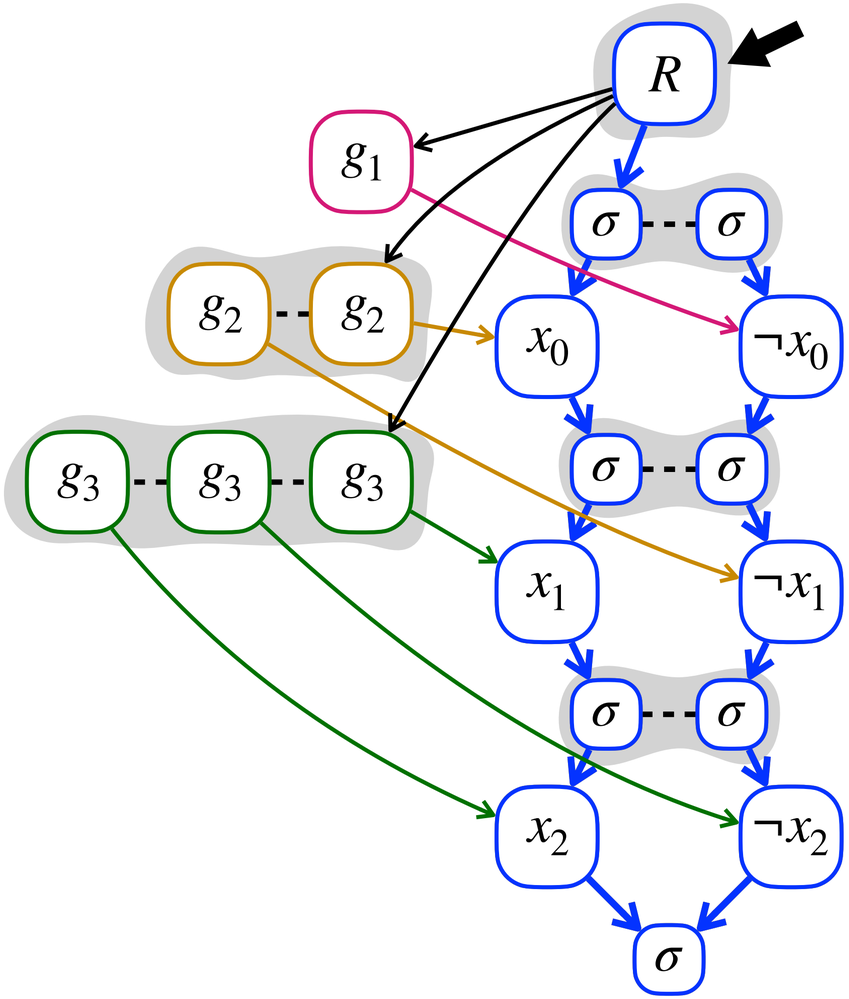


Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$



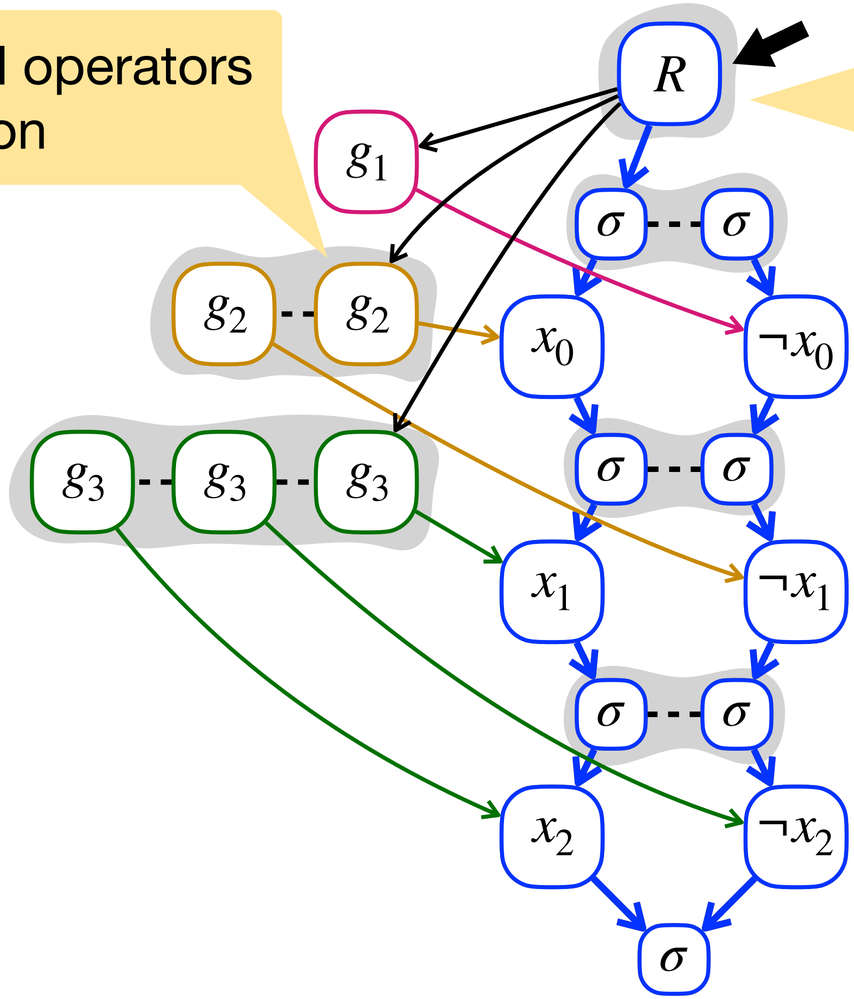
Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

Terms relying on effectful operators encode clause satisfaction



Extracted term ensures valid assignment and satisfaction of clauses

Effectful operators encode variable assignments

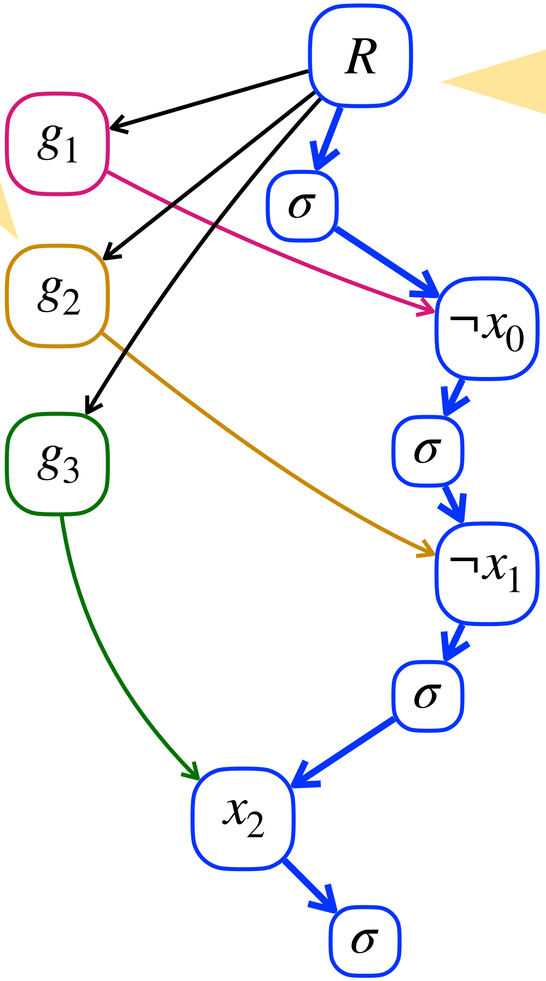
Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

Terms relying on effectful operators encode clause satisfaction



Extracted term ensures valid assignment and satisfaction of clauses

Effectful operators encode variable assignments

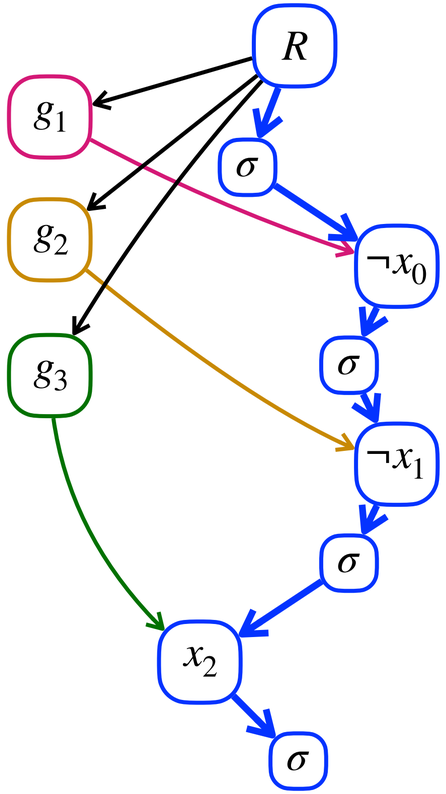
Proof Strategy

1. Take a boolean formula ϕ.
2. Construct an e-graph $\mathcal{G}$.
3. Show that $\mathcal{G}$ has an effect-safe extraction if and only if ϕ is satisfiable.

Effect-safe Extraction is NP-Hard

$$\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$$

Effect-Safe Extraction from $\mathcal{G}$



Satisfying Assignment of ϕ:

$x_0 = \bot$

$x_1 = \bot$

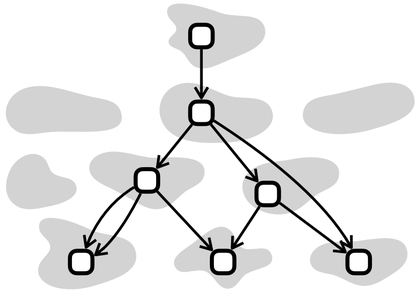
$x_2 = \top$

NP-Hard ⇒ Give Up?

NP-Hard ⇒ ~~Give Up?~~

NP-Hard ⇒ Try a solver

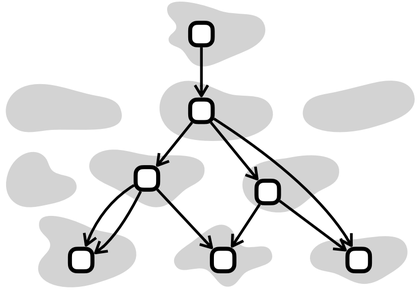
Solver-Based Effect-Safe Extraction

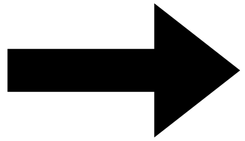


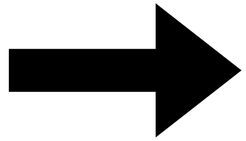
Set of linear equations

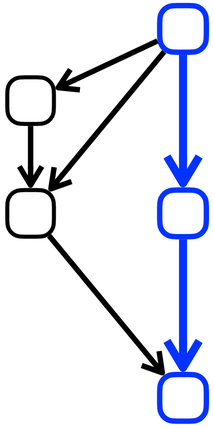
- No cycles
- Well-formed term
- Linearity constraints
- Cost is minimized
- …

Solver-Based Effect-Safe Extraction





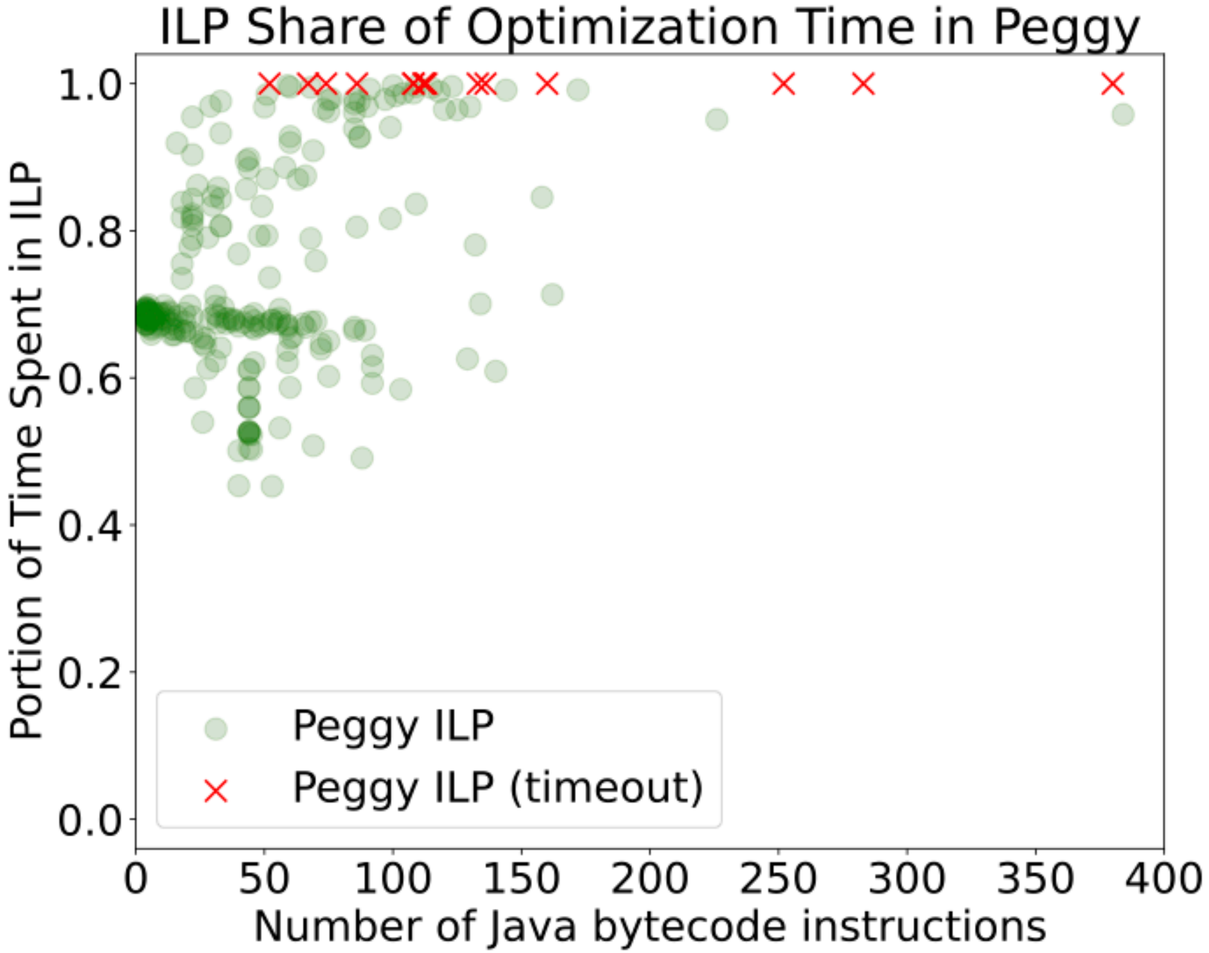




Set of linear equations

- No cycles
- Well-formed term
- Linearity constraints
- Cost is minimized
- …

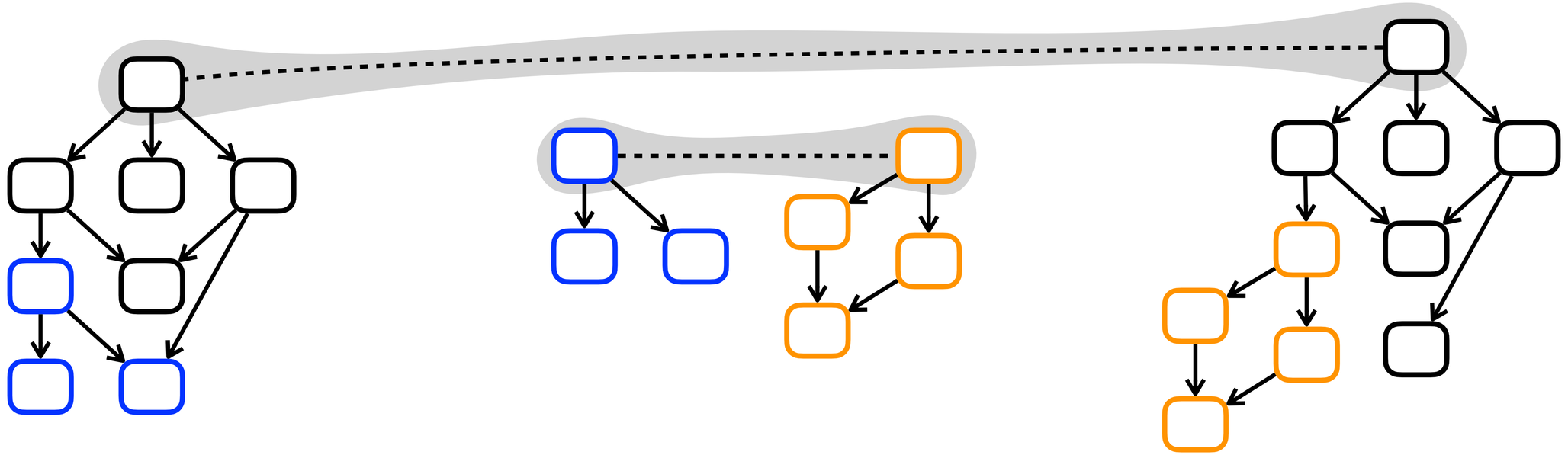
Solver-Based Effect-Safe Extraction



ILP-Based Extraction Bottleneck

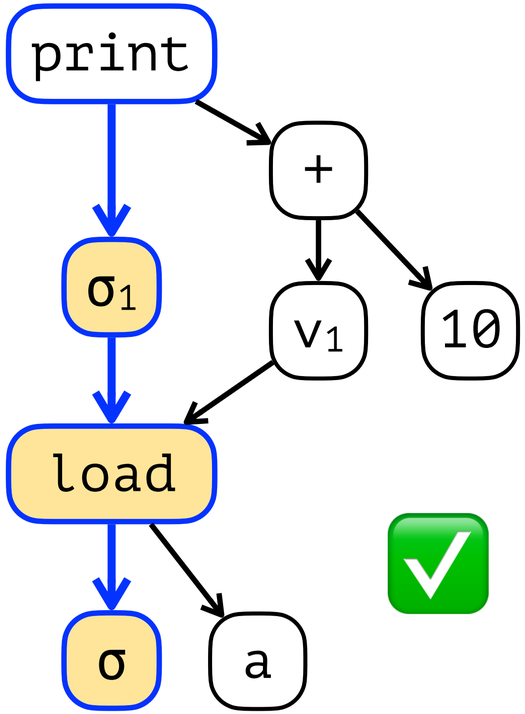
NP-Hard ⇒ Find a parameterization that is fast in practice

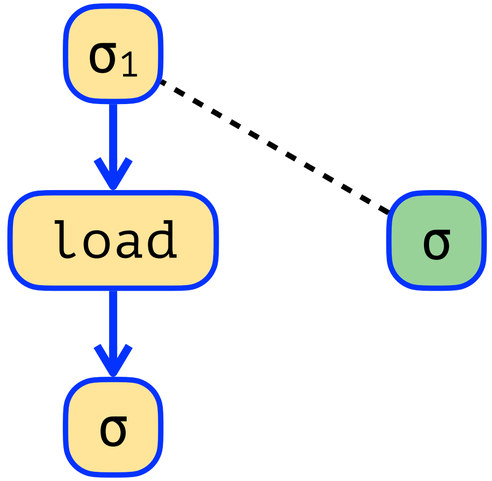
Fast Effect-safe Extraction in Practice

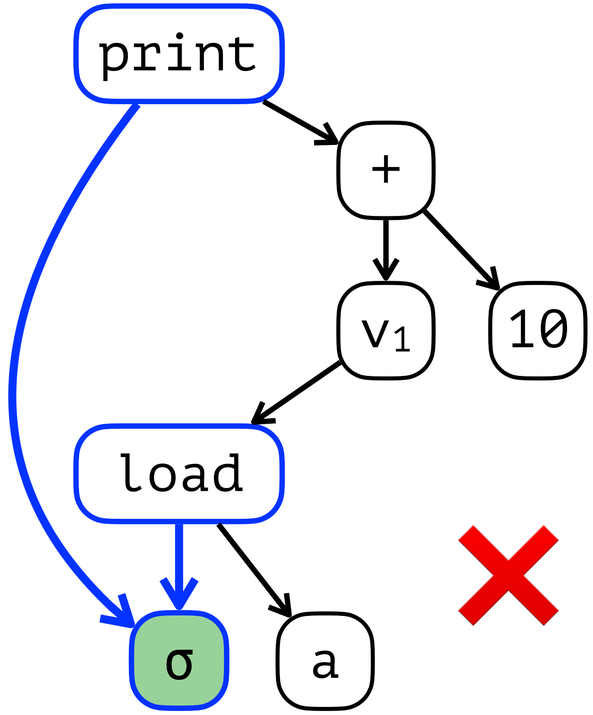


Greedy extraction relied on interchangeability of equivalent terms

Fast Effect-safe Extraction in Practice

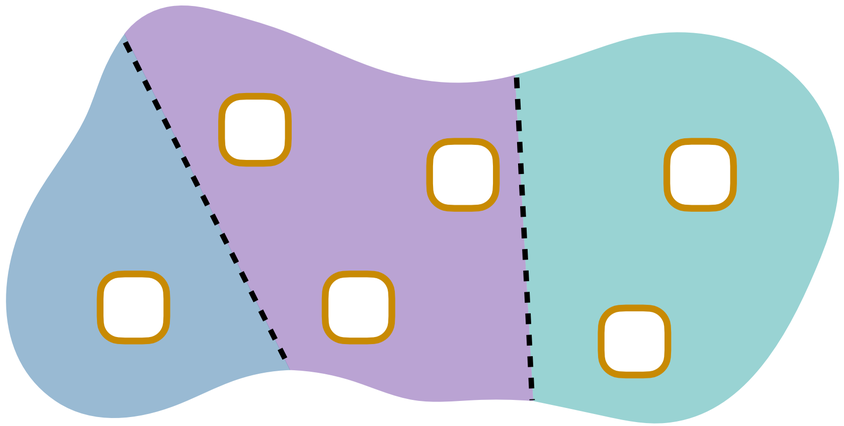






Effectful terms are <u>not</u> interchangeable

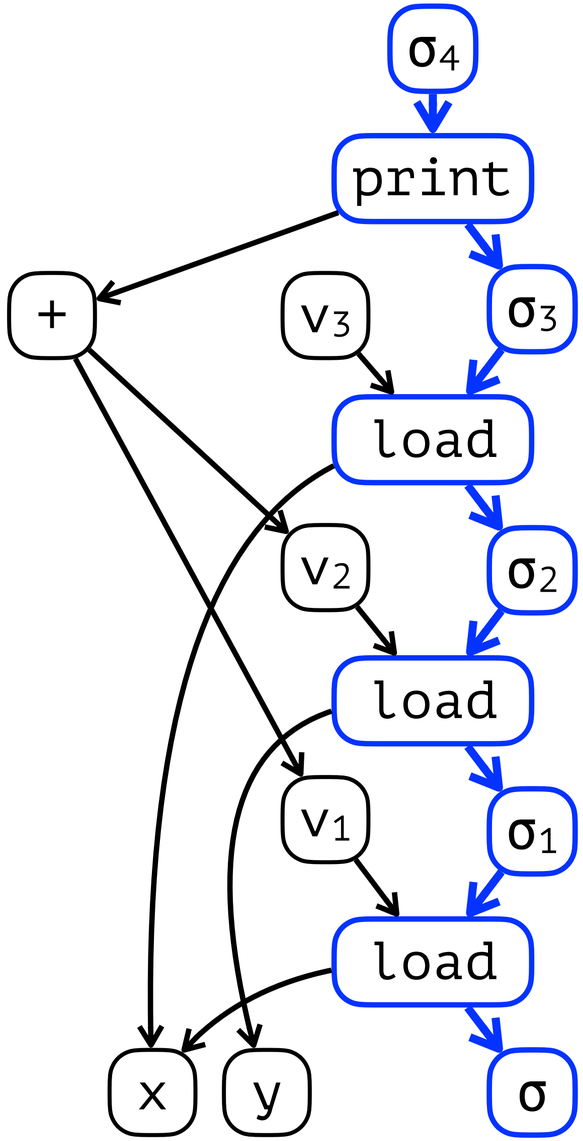
Key Idea: Refine the Equivalence Relation



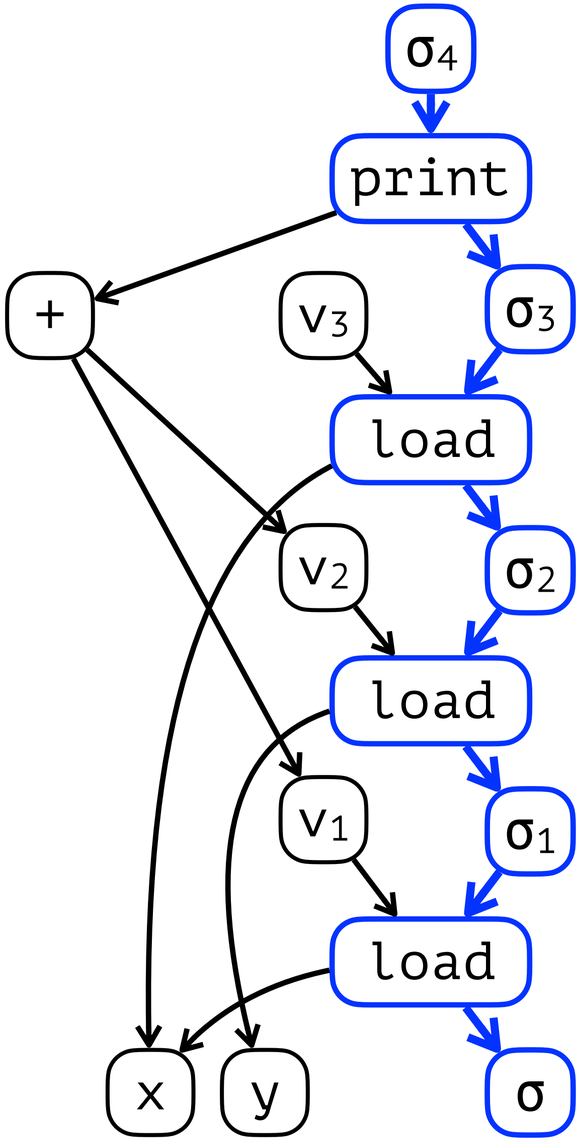
Group effectful terms by the values they compute

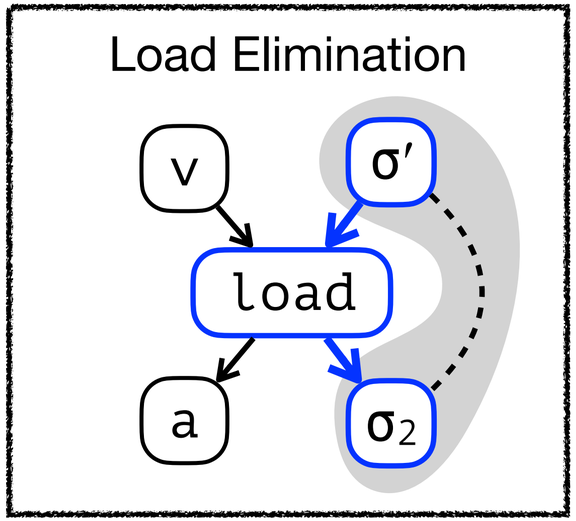
```
(v1, σ1) = load(x)
(v2, σ2) = load(y)
(v3, σ3) = load(x)
     σ4  = print(v1 + v2)
```

```
(v1, σ1) = load(x)
(v2, σ2) = load(y)
(v3, σ3) = load(x)
     σ4  = print(v1 + v2)
```

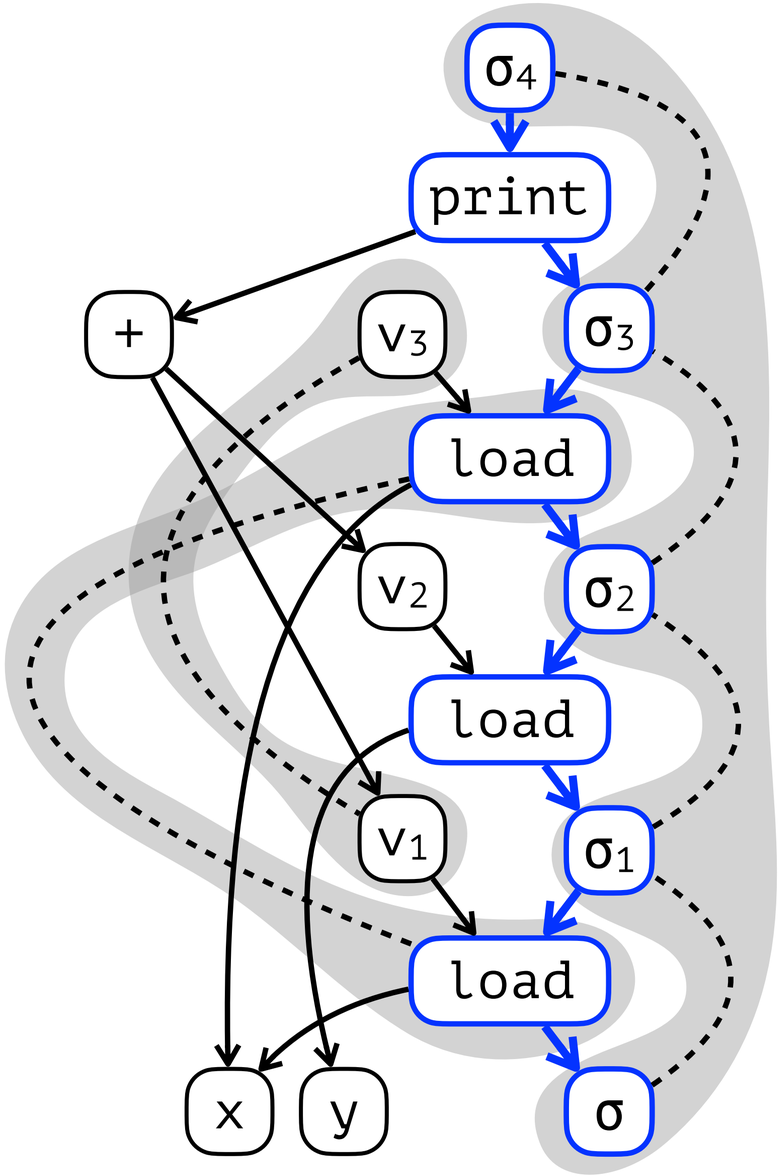


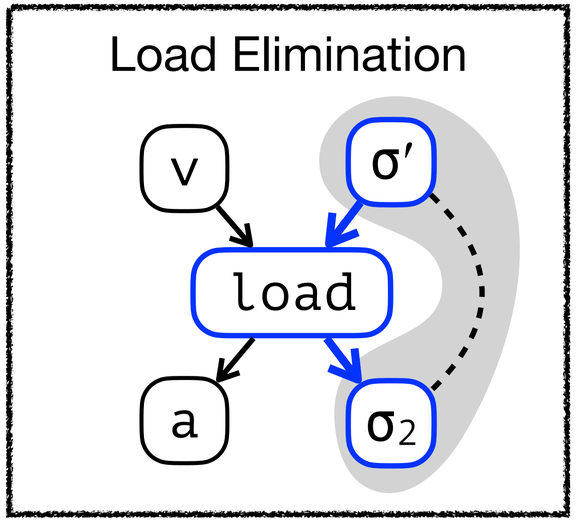
```
(v1, σ1) = load(x)
(v2, σ2) = load(y)
(v3, σ3) = load(x)
     σ4  = print(v1 + v2)
```

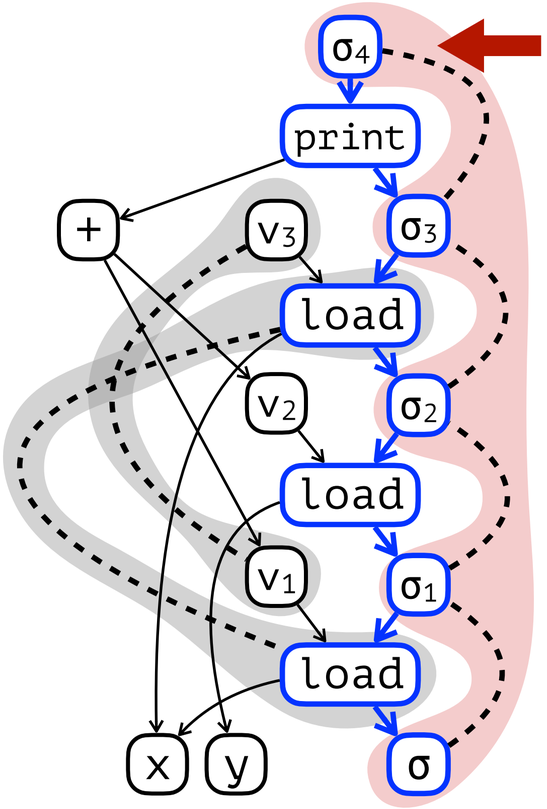


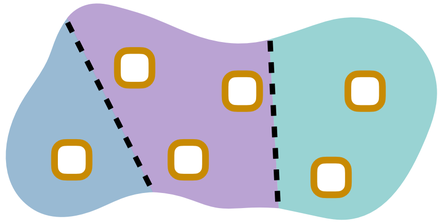


```
(v1, σ1) = load(x)
(v2, σ2) = load(y)
(v3, σ3) = load(x)
     σ4  = print(v1 + v2)
```

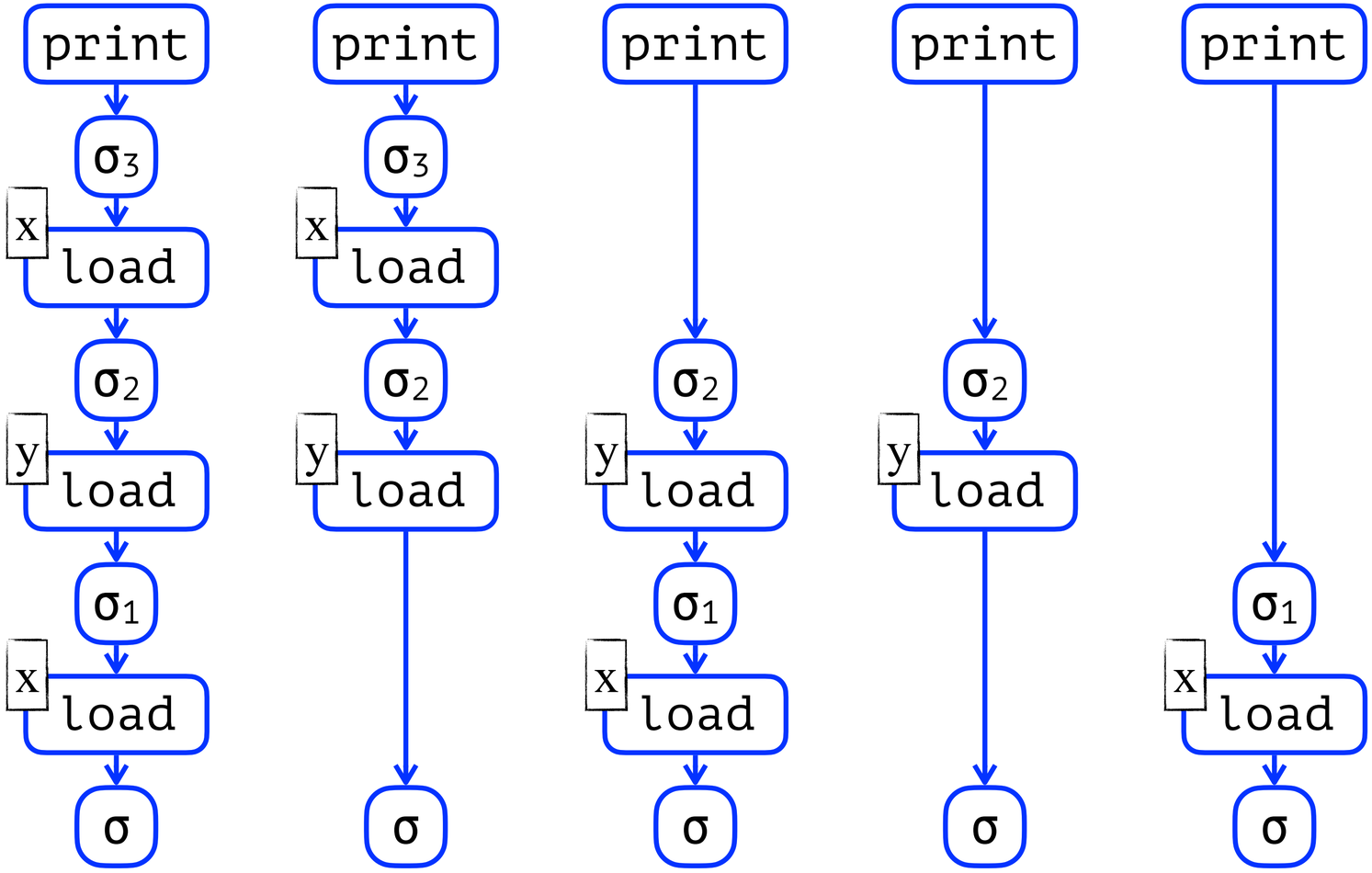


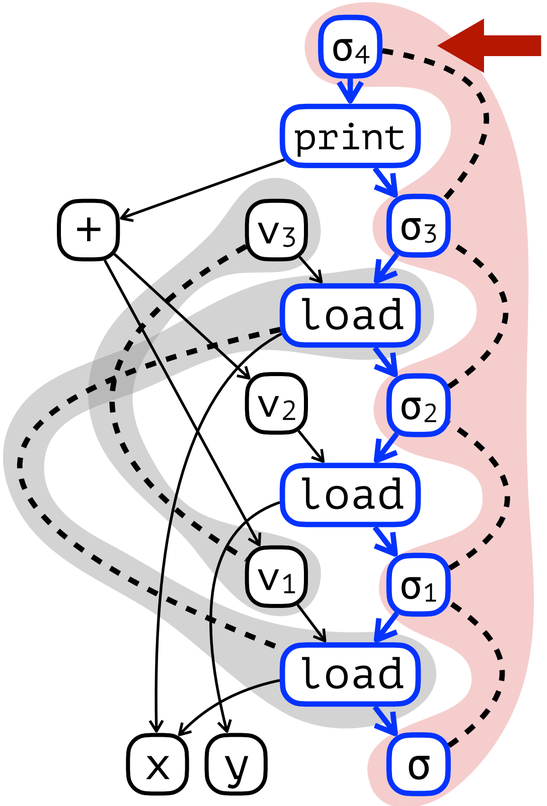


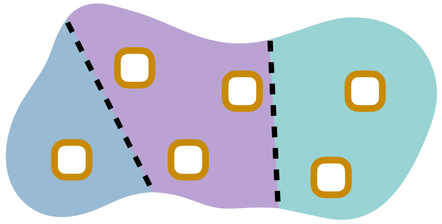




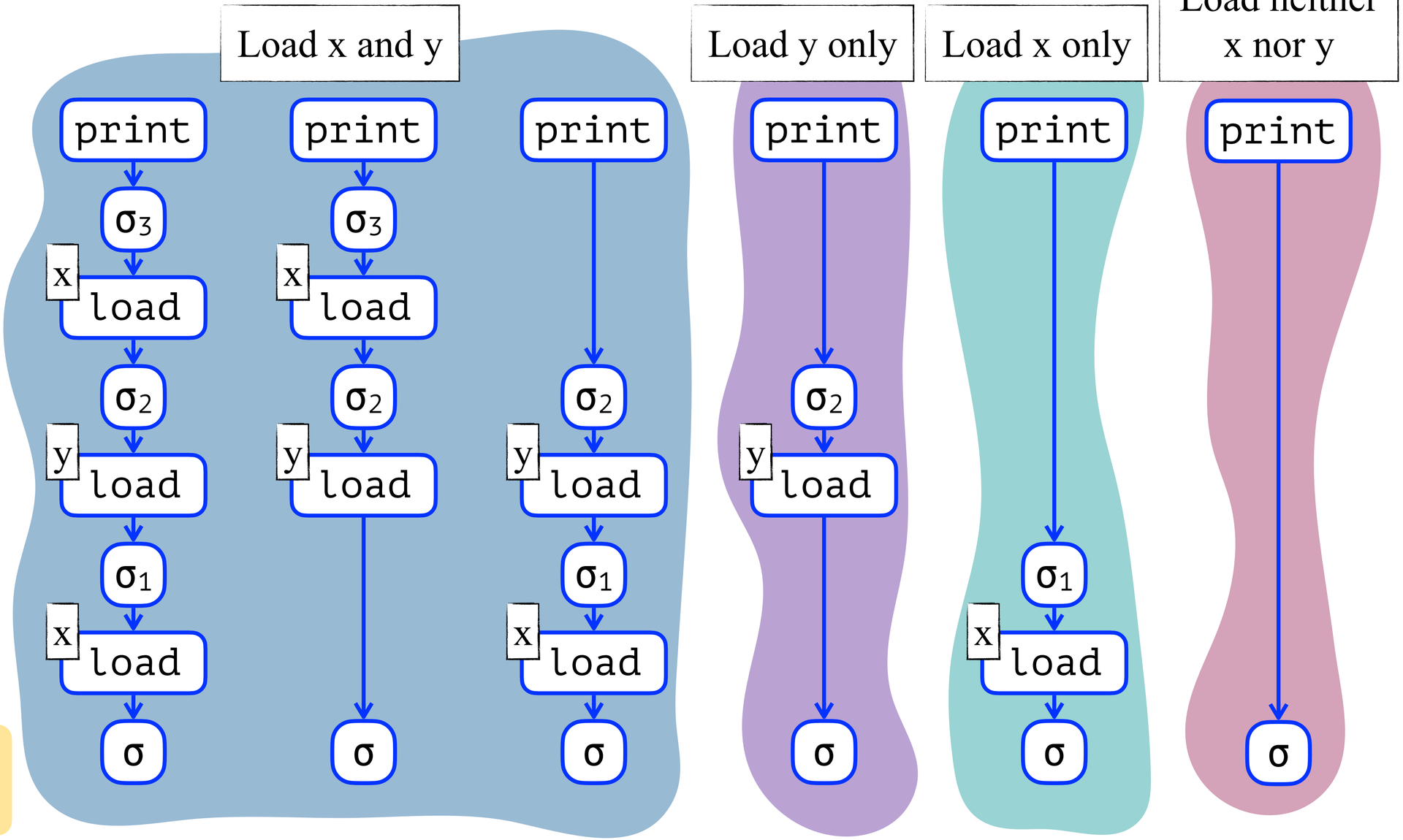
Group effectful terms by the values they compute



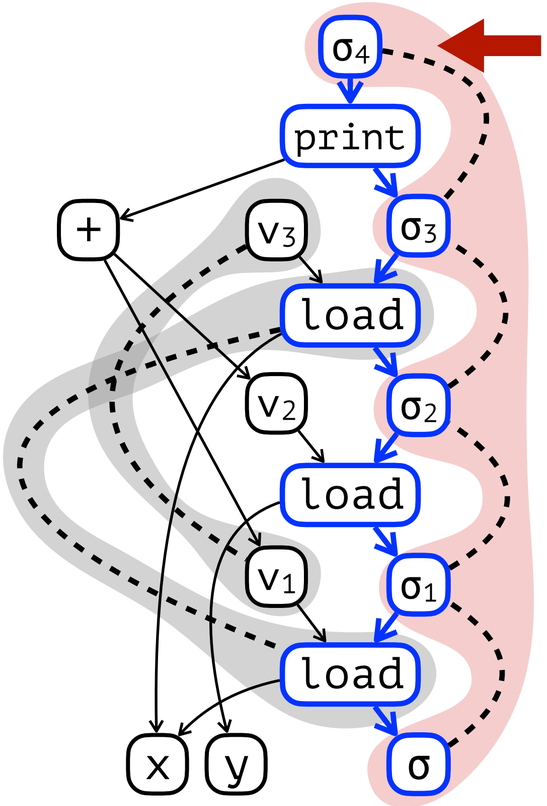


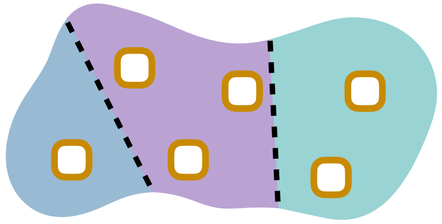


Group effectful terms by the values they compute

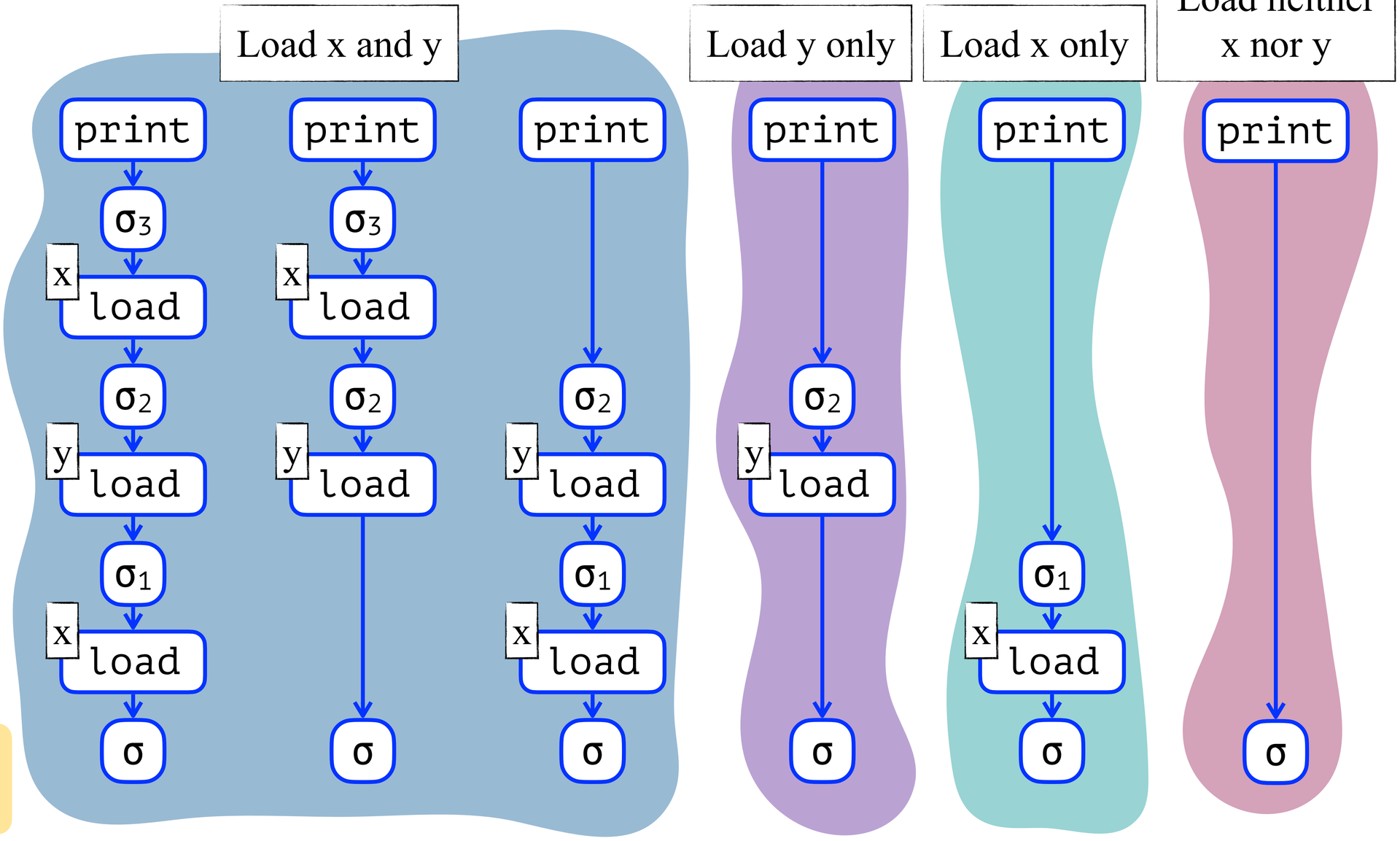


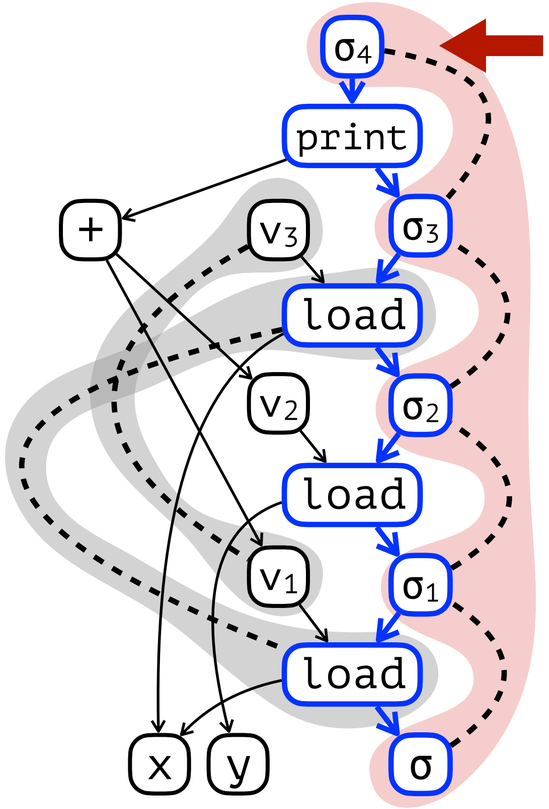
Terms are interchangeable within each partition

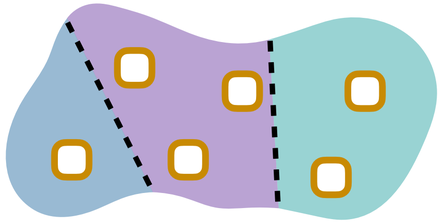




Group effectful terms by the values they compute



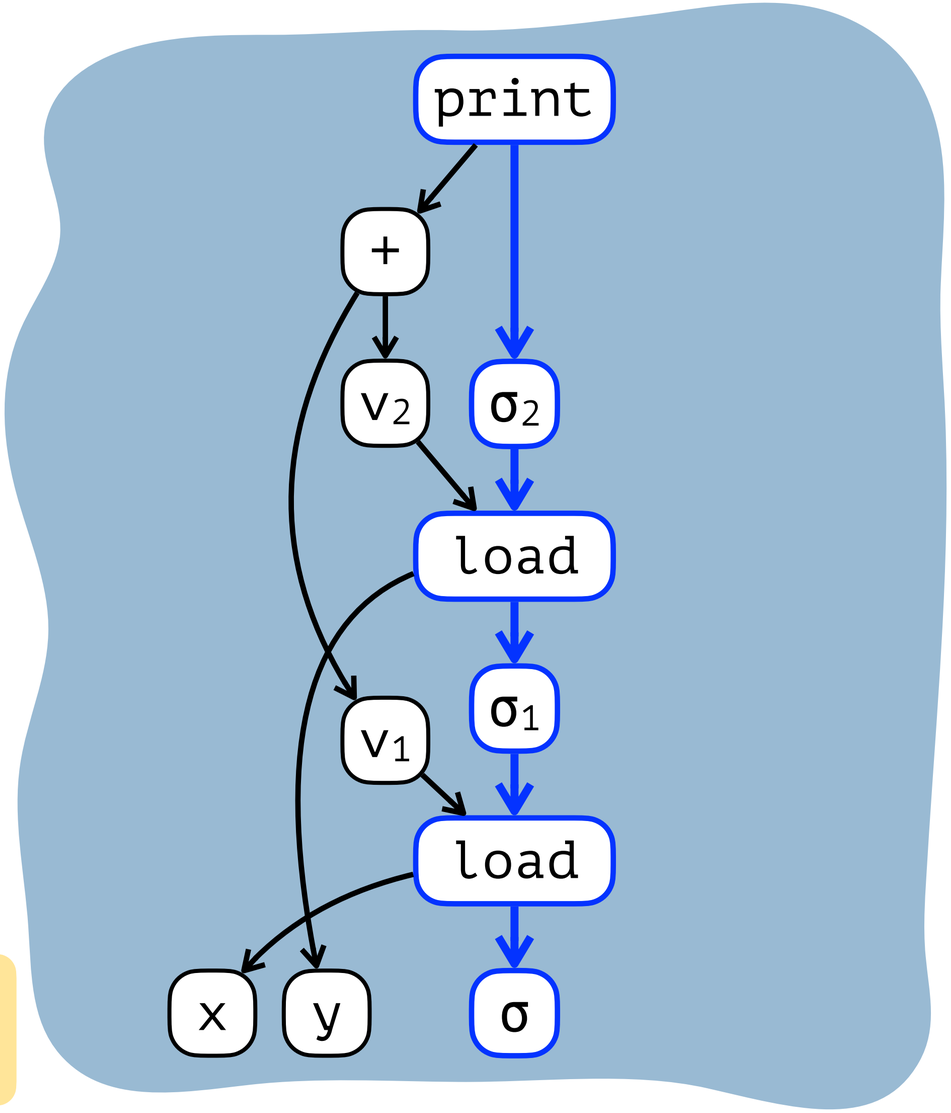


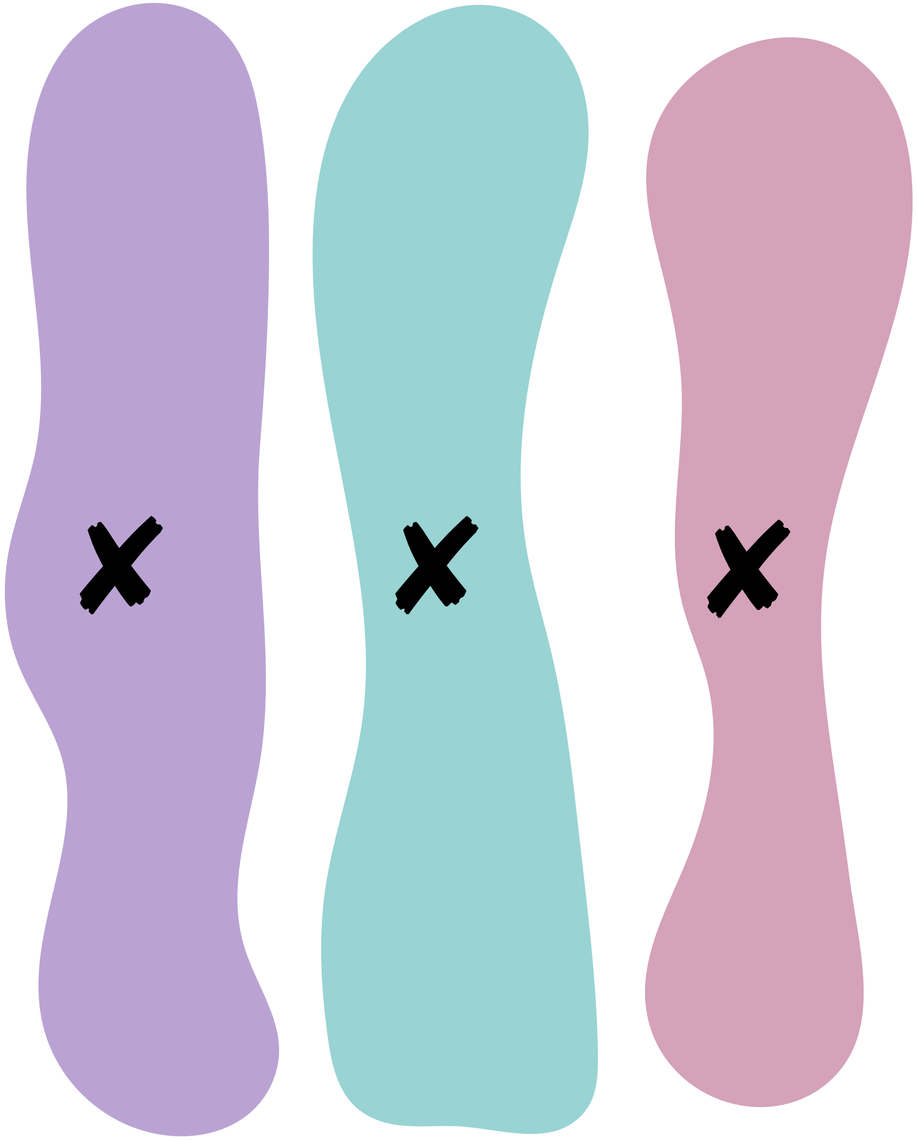


Group effectful terms by the values they compute

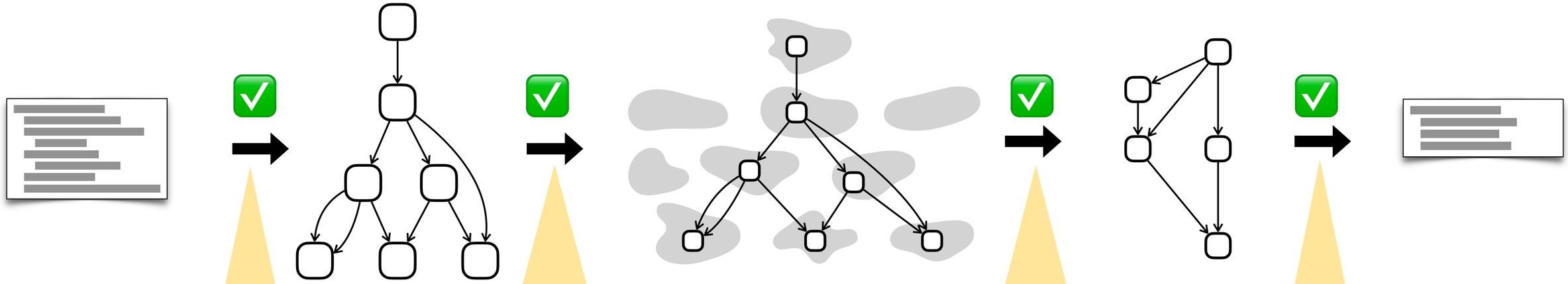
Terms are interchangeable within each partition

Find one low-cost extraction for each





Effectful Program Optimization with E-Graphs



Encode order of effects with explicit state values

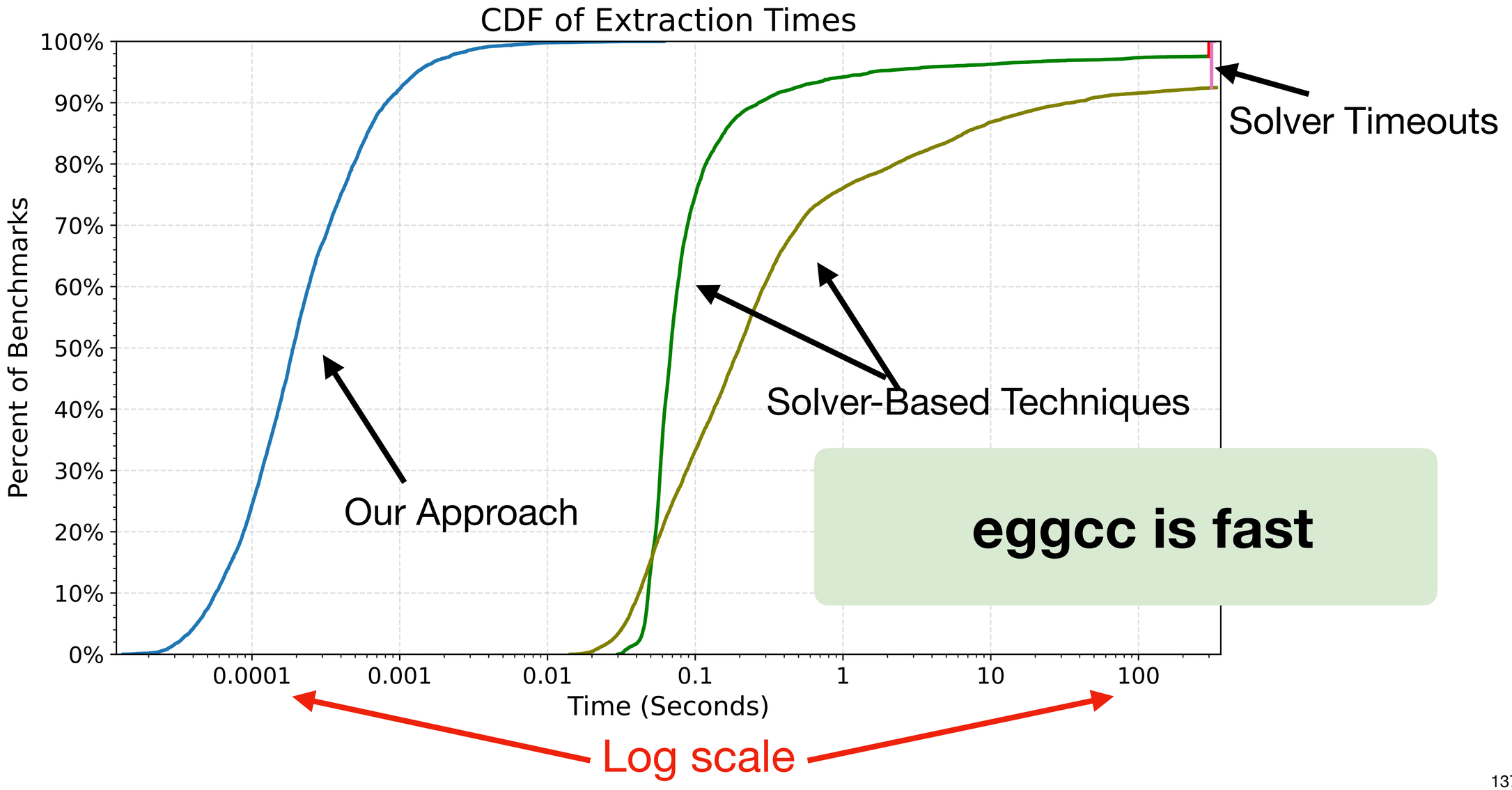
Relaxed equivalence relation for optimization

Effect-safe extraction under refined equivalence relation

Reconstruction succeeds if effectful operations form a linear sequence

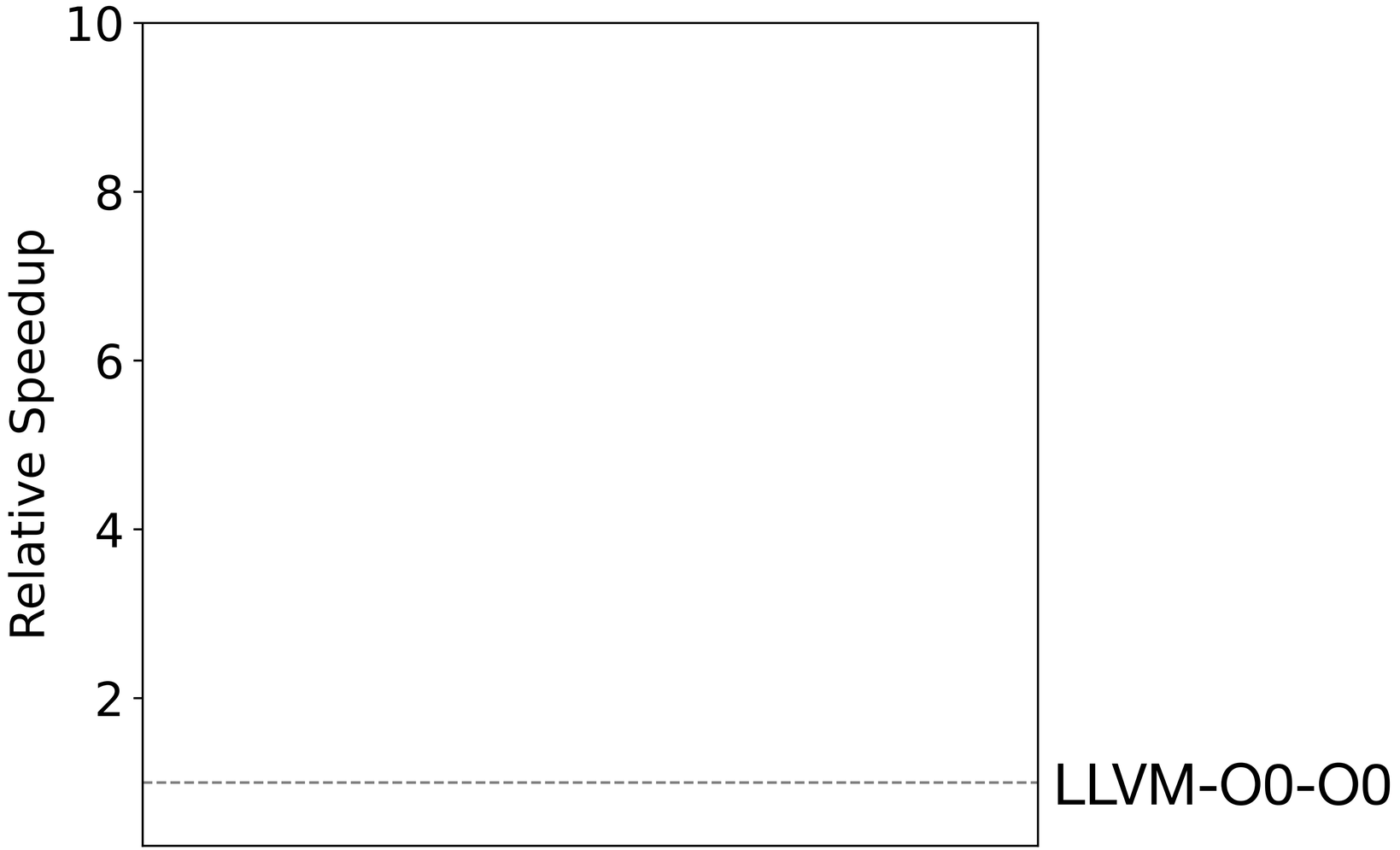
The eggcc compiler

Fast Effect-safe Extraction in Practice

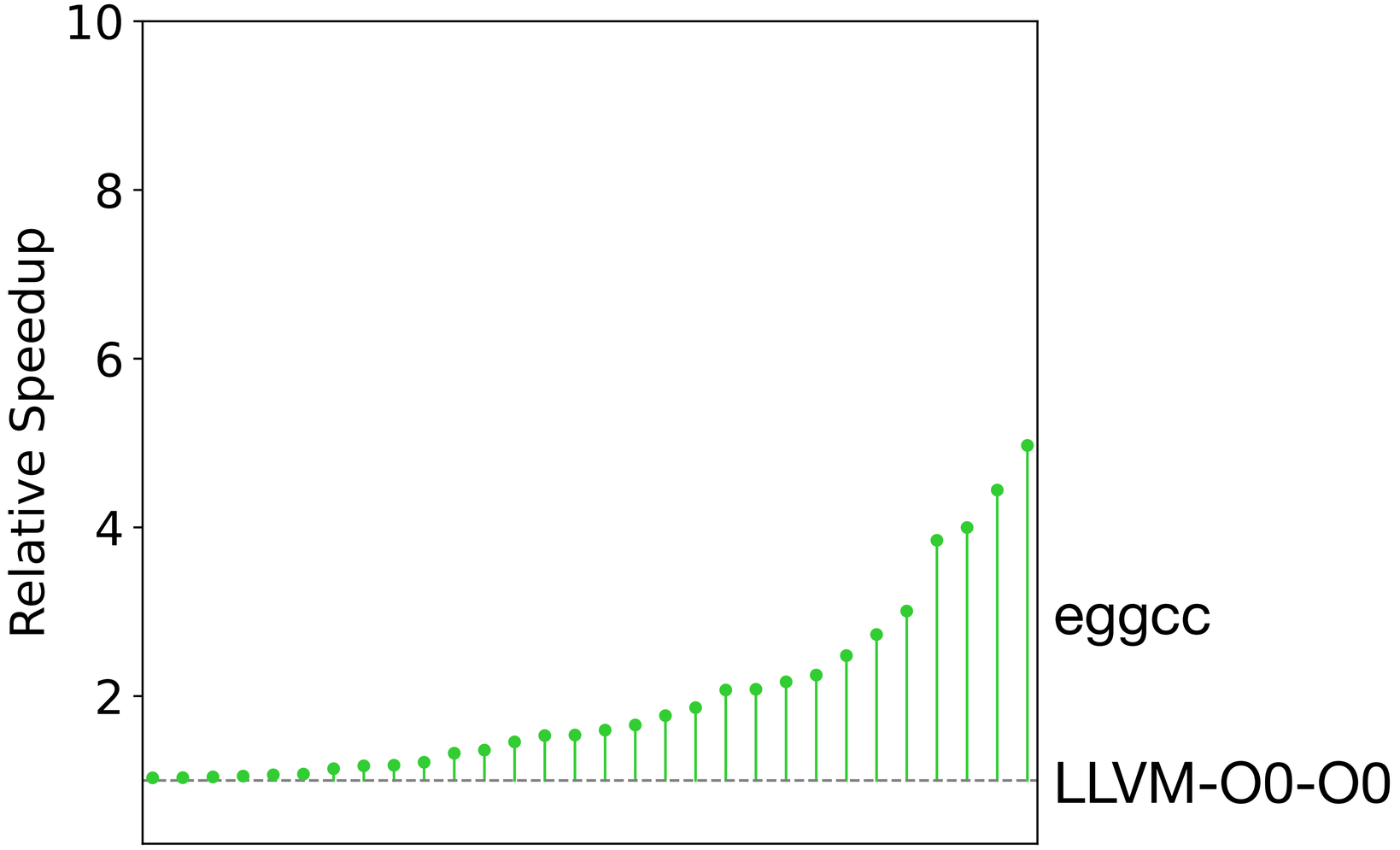


eggcc is fast

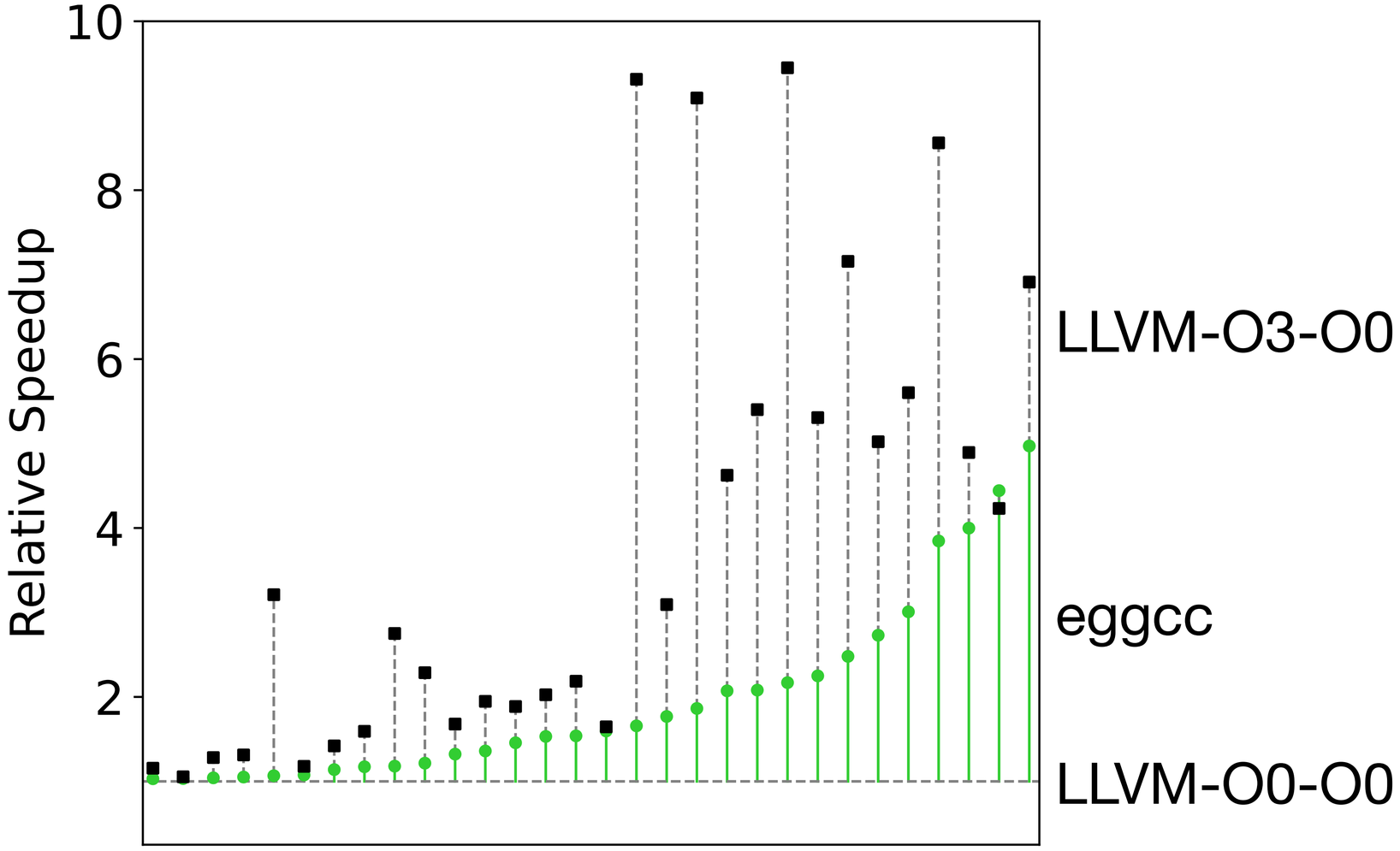
Fast Effect-safe Extraction in Practice



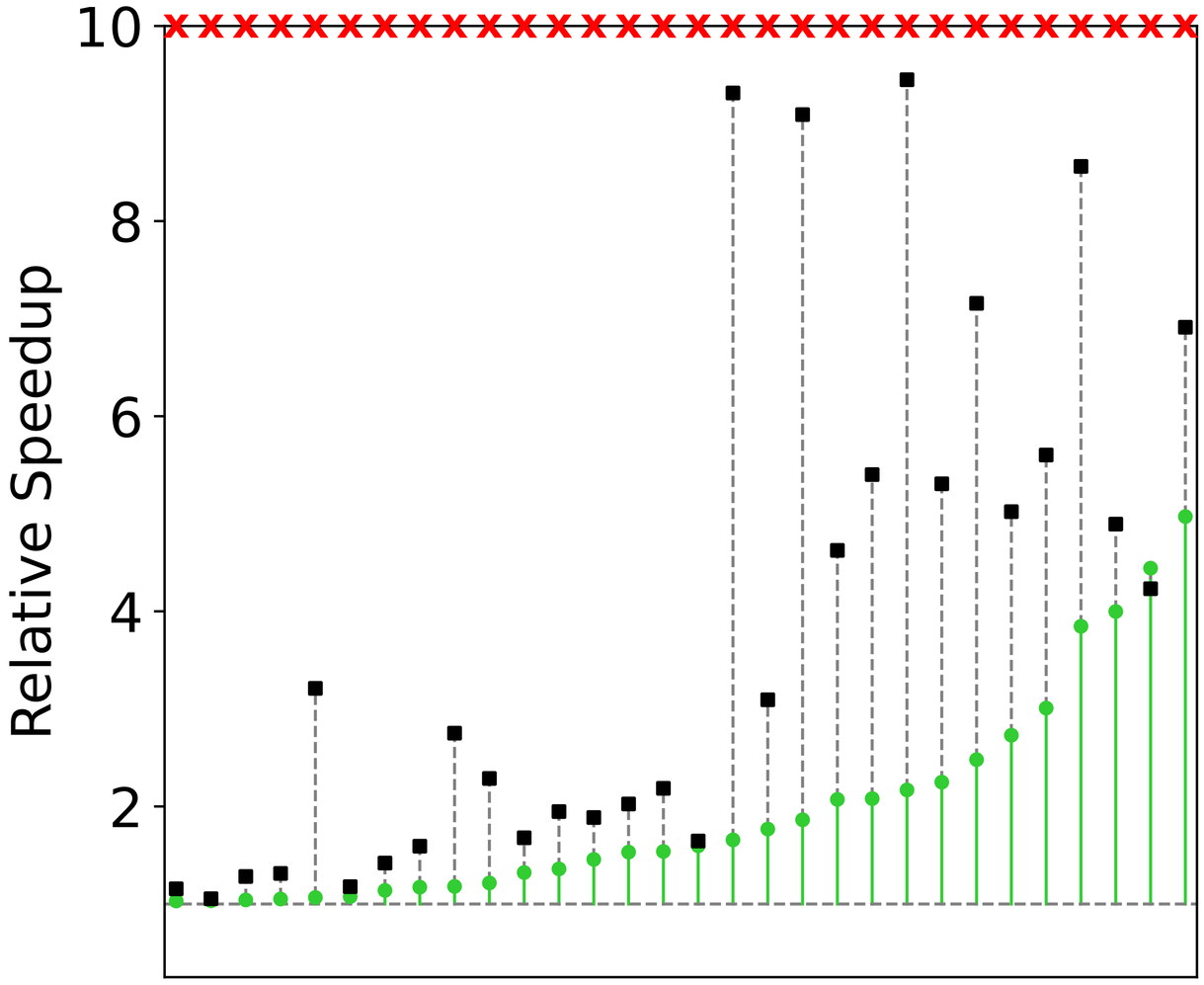
Fast Effect-safe Extraction in Practice



Fast Effect-safe Extraction in Practice



Fast Effect-safe Extraction in Practice



Solver Approaches

LLVM-O3-O0

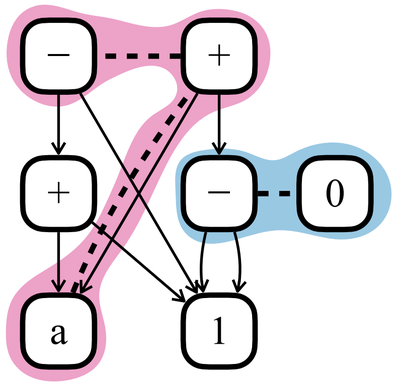
eggcc

LLVM-O0-O0

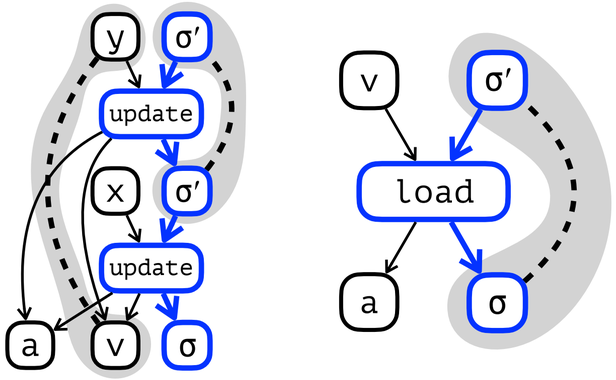
eggcc finds good output programs

Fast Effect-safe Extraction in Practice

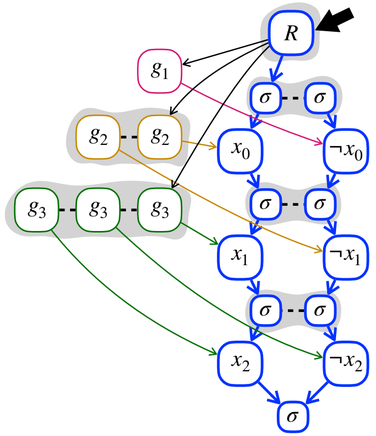
Rewrite non-destructively with e-graphs



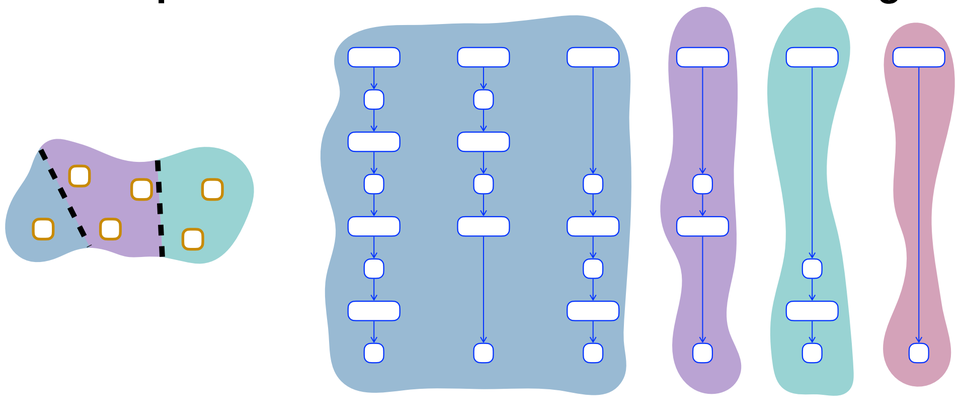
Relax equivalence relation to enable optimizations



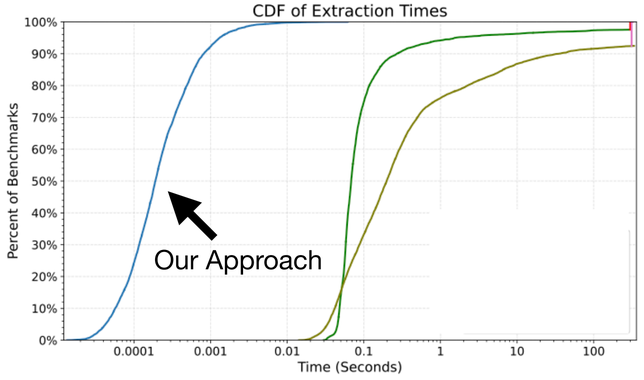
Effect-safe extraction is NP-Complete

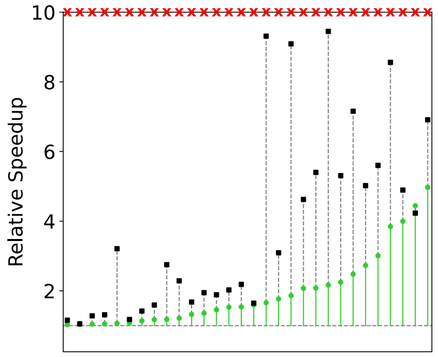


Refine equivalence relation to recover interchangeability

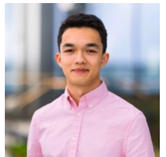


Empirically fast, high-quality extraction













PLSE