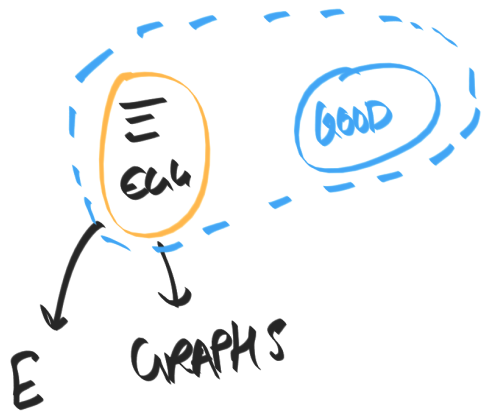
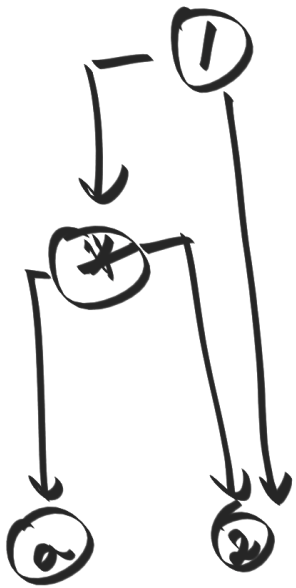
egg : Fast and extensible equality saturation

Siddharth Bhat

Monday, Jan 18 2021

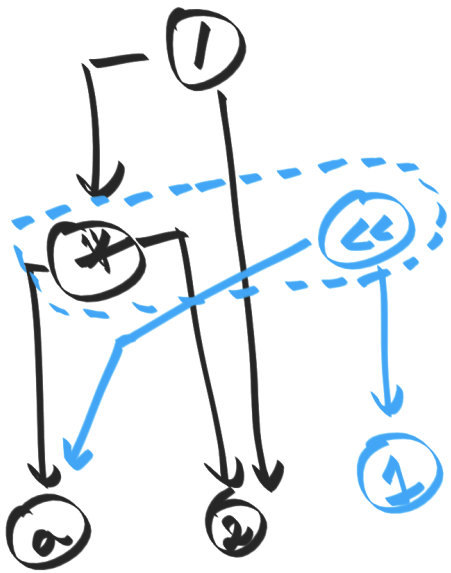


egg : Fast and extensible equality saturation



(a*2)/2 :
(1 (* a 2) 2

egg : Fast and extensible equality saturation (2)

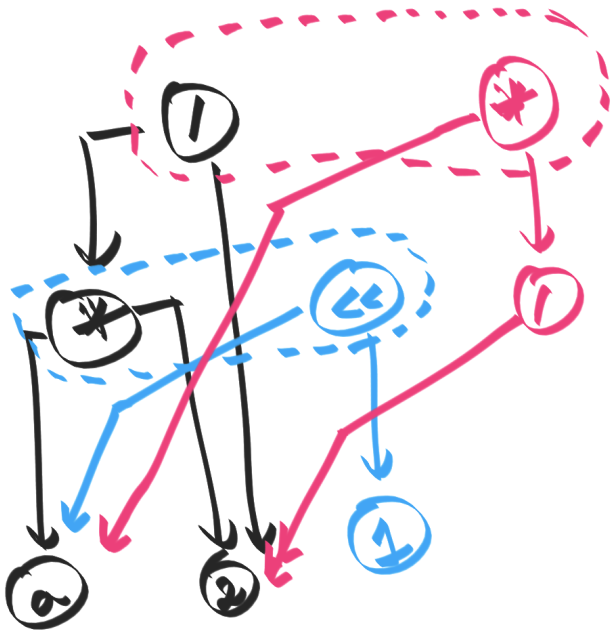


(a*2)/2:
(/ (* a 2) 2)

p*2 → p<<1

(/ (* a 2) 2)
(<< a 1)

egg : Fast and extensible equality saturation (3)



(a*2)/2:
(/ (* a 2) 2)

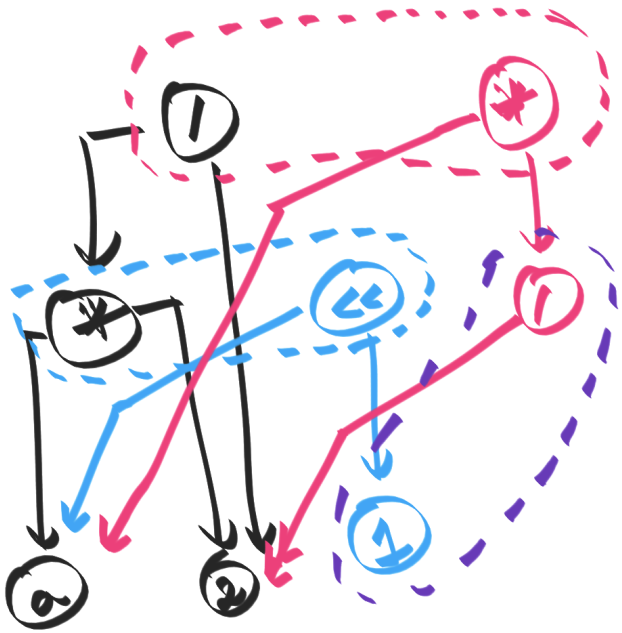
p*2 → p<<1

(/ (* a 2) 2)
(<< a 1)

(p*a)/2 → p*(a/2)

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

egg : Fast and extensible equality saturation (4)



$x/x \to 1$

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

$(a*2)/2:$
$(/ (* a 2) 2)$

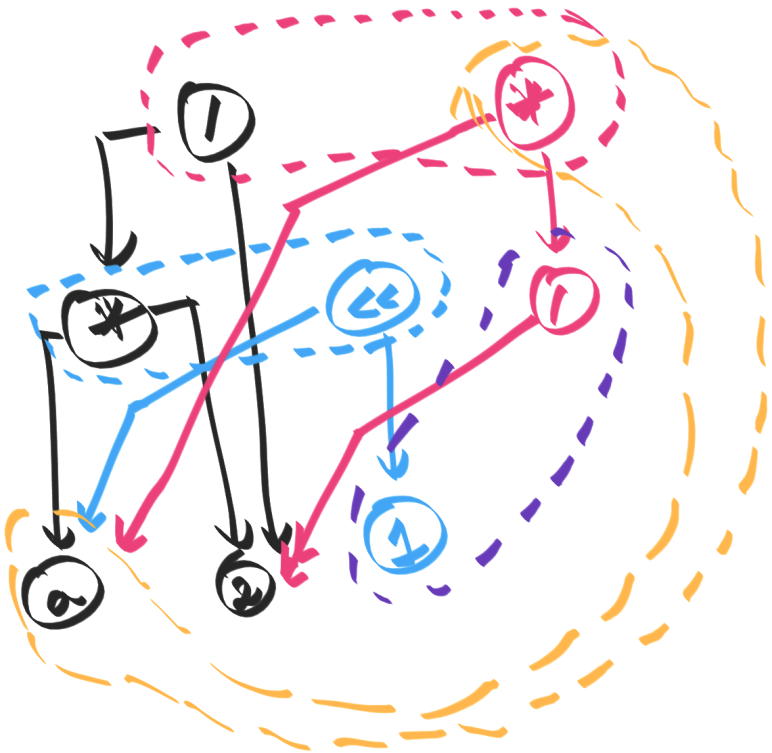
$p*2 \to p<<1$

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

$(p*a)/2 \to p*(a/2)$

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

egg : Fast and extensible equality saturation (5)



$x/x \to 1$

$(* a (/ 2 2))$

$x * 1 \to x$

$(* a (/ 2 2))$

a

$(a*2)/2:$

$(/ (* a 2) 2)$

$p*2 \to p << 1$

$(/ (* a 2) 2)$

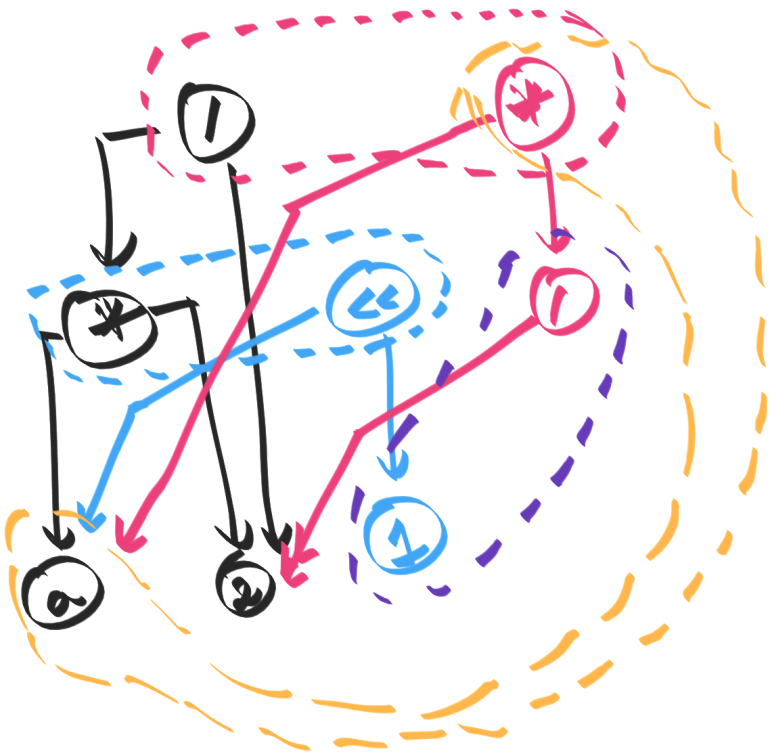
$(<< a 1)$

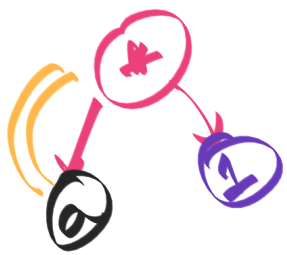
$(p*a)/x \to p*(a/x)$

$(/ (* a 2) 2)$

$(* a (/ 2 2))$

egg : Fast and extensible equality saturation (6)





x/x → 1

(* a (/ 2 2))

1

x*1 → x

(* a (/ 2 2))

1

a

(a*2)/2 :

(/ (* a 2) 2)

p*2 → p<<1

(/ (* a 2) 2)

(<< a 1)

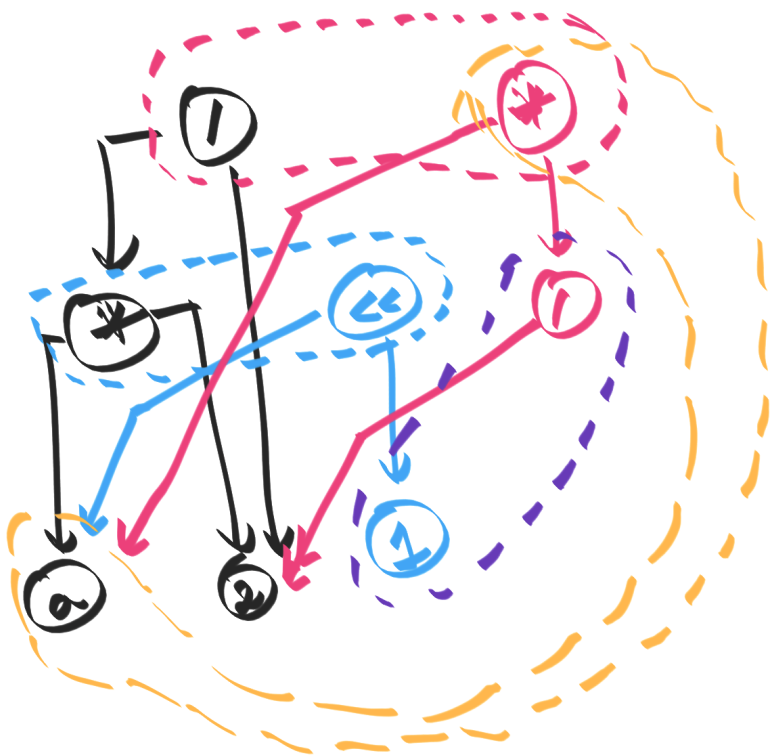
(p*a)/x → p*(a/x)

(/ (* a 2) 2)

2

(* a (/ 2 2))

egg : Fast and extensible equality saturation (7)



$x/x \to 1$

$(* a (/ 2 2))$

$(* a 1)$

$x * 1 \to x$

$(* a (/ 2 2))$

$(* a 1)$

a

$(a*2)/2$:

$(/ (* a 2) 2)$

$p * 2 \to p << 1$

$(/ (* a 2) 2)$

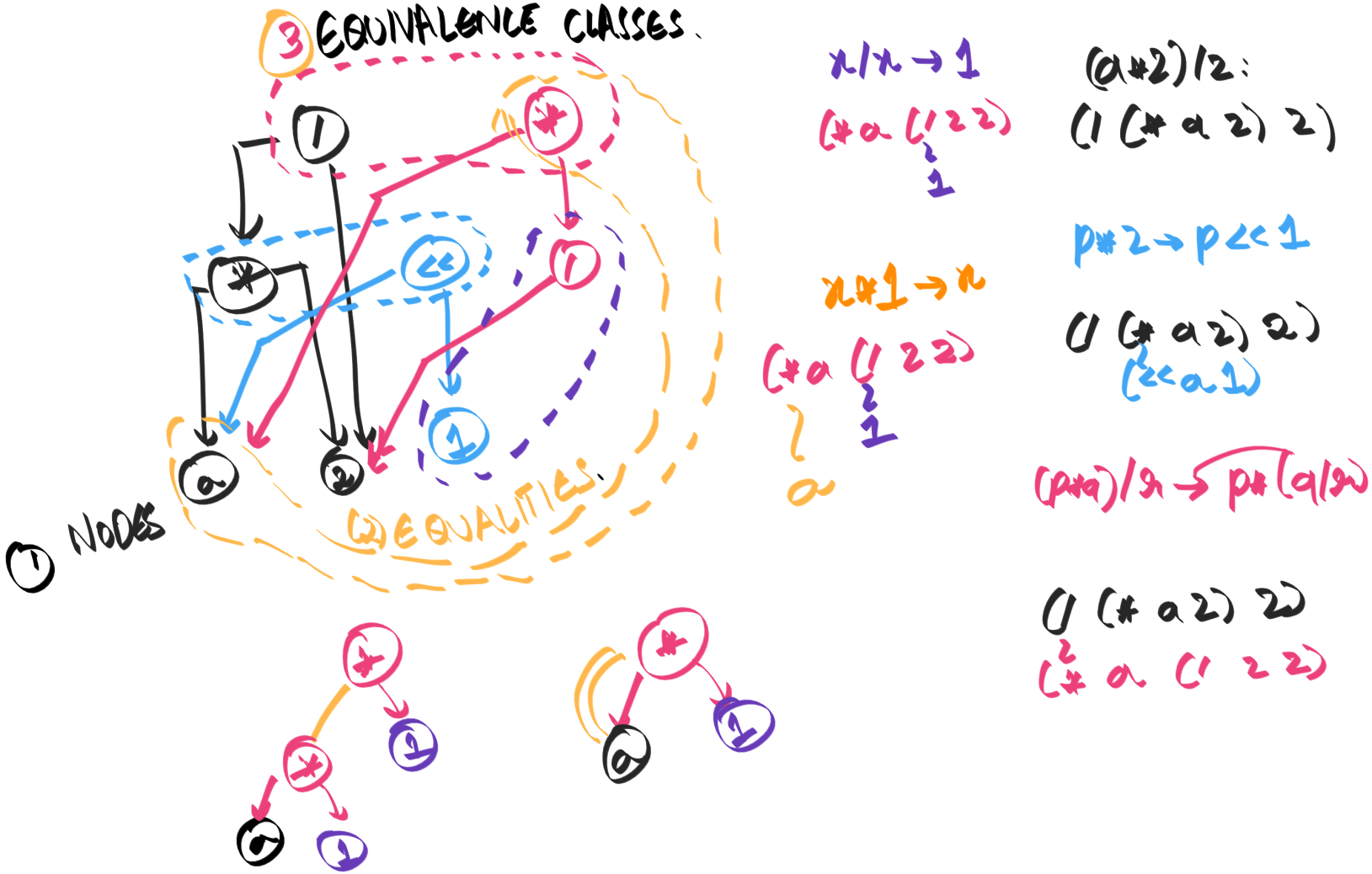
$(<< a 1)$

$(p*q)/2 \to p*(q/2)$

$(/ (* a 2) 2)$

$(* a (/ 2 2))$

egg : Fast and extensible equality saturation (8)



Evaluation: Herbie

`sqrt(x+1) - sqrt(x)` → `1/(sqrt(x+1) + sqrt(x))`

Herbie detects inaccurate expressions and finds more accurate replacements. The red expression is inaccurate when $x > 1$; Herbie's replacement, in blue, is accurate for all x.

Evaluation: Herbie

`sqrt(x+1) - sqrt(x)` → `1/(sqrt(x+1) + sqrt(x))`

Herbie detects inaccurate expressions and finds more accurate replacements. The red expression is inaccurate when $x > 1$; Herbie's replacement, in blue, is accurate for all x.

$$\sqrt{x+1}-\sqrt{x} = \frac{(\sqrt{x+1}-\sqrt{x})(\sqrt{x+1}+\sqrt{x})}{\sqrt{x+1}+\sqrt{x}} = \frac{(x+1)-x}{\sqrt{x+1}+\sqrt{x}} = \frac{1}{(\sqrt{x+1}+\sqrt{x})}$$

Evaluation: Herbie

`sqrt(x+1) - sqrt(x)` → `1/(sqrt(x+1) + sqrt(x))`

Herbie detects inaccurate expressions and finds more accurate replacements. The red expression is inaccurate when $x > 1$; Herbie's replacement, in blue, is accurate for all x.

$$\sqrt{x+1}-\sqrt{x} = \frac{(\sqrt{x+1}-\sqrt{x})(\sqrt{x+1}+\sqrt{x})}{\sqrt{x+1}+\sqrt{x}} = \frac{(x+1)-x}{\sqrt{x+1}+\sqrt{x}} = \frac{1}{(\sqrt{x+1}+\sqrt{x})}$$

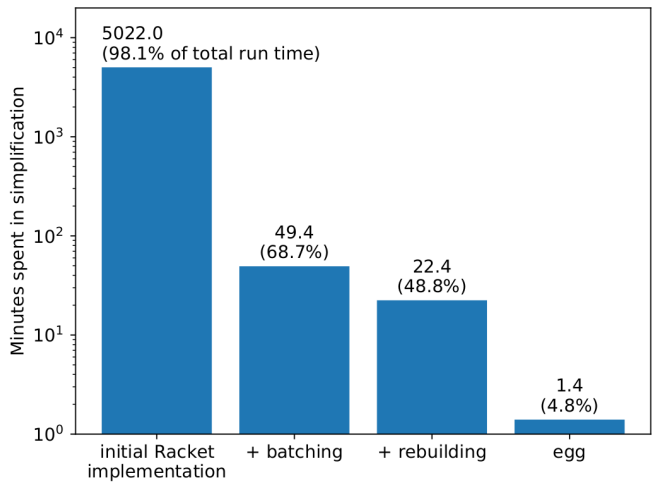
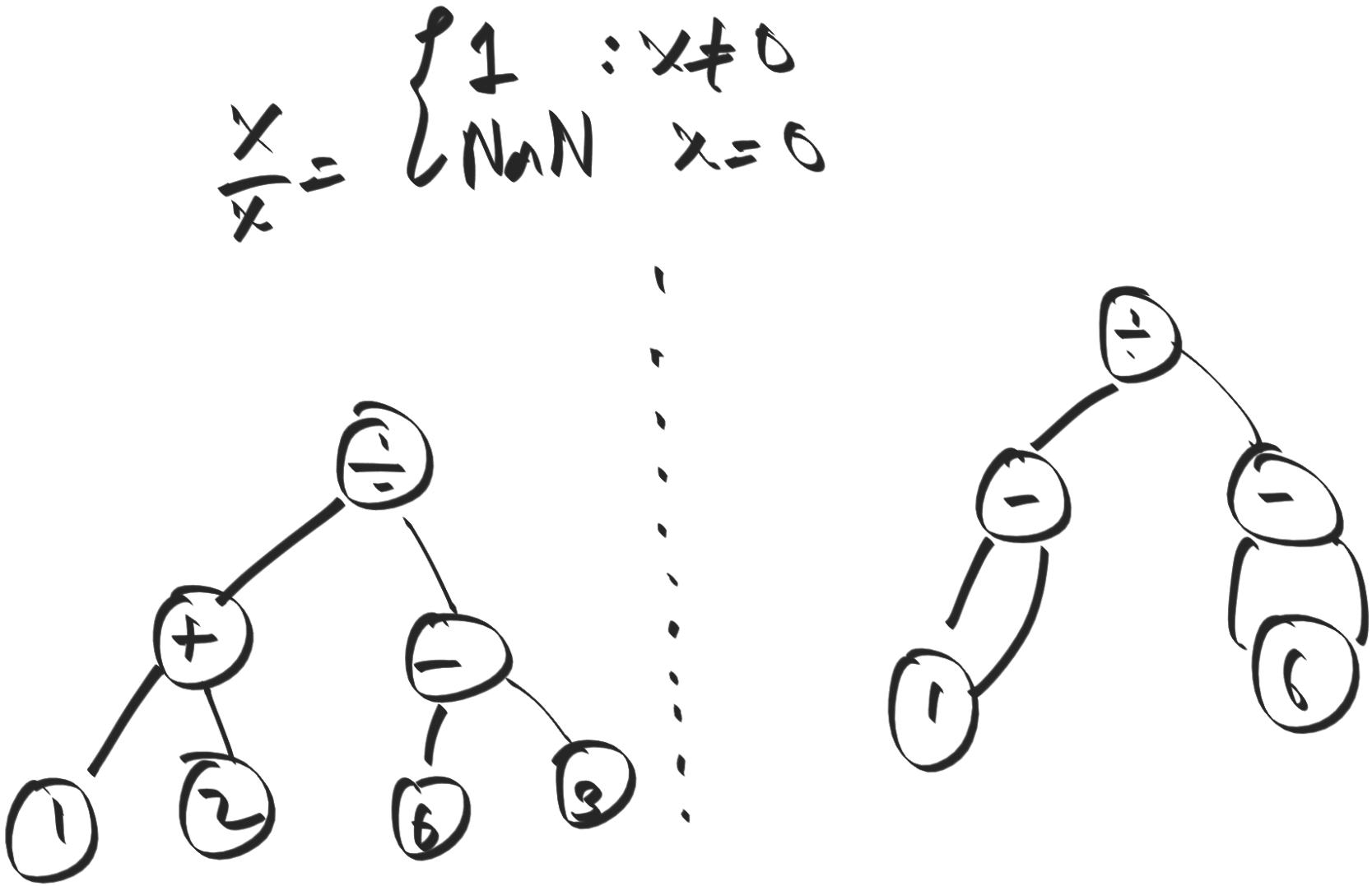
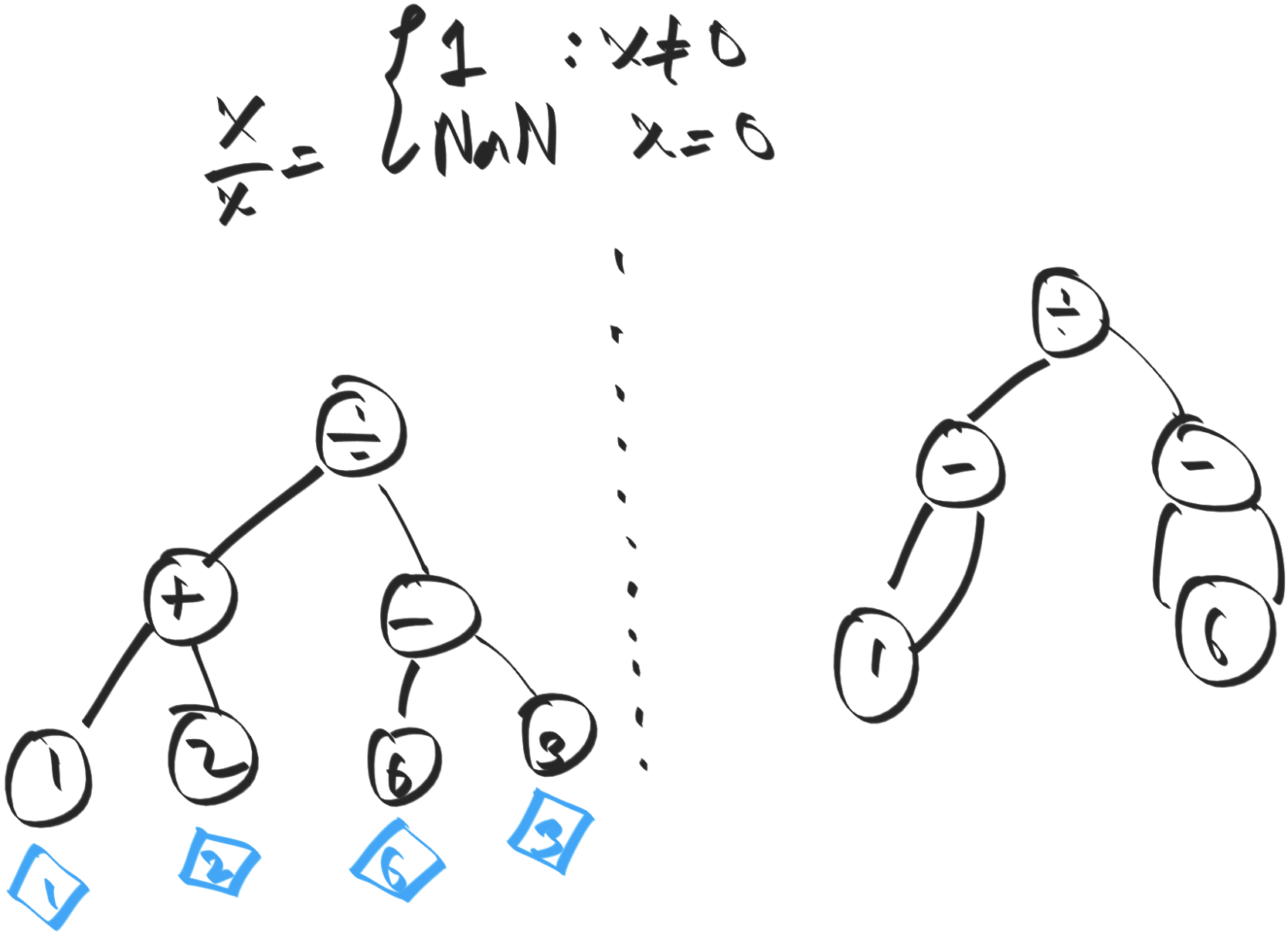


Fig. 12. Herbie sped up its expression simplification phase by adopting egg-inspired features like batched simplification and rebuilding into its Racket-based e-graph implementation. Herbie also supports using egg itself for additional speedup. Note that the y-axis is log-scale.

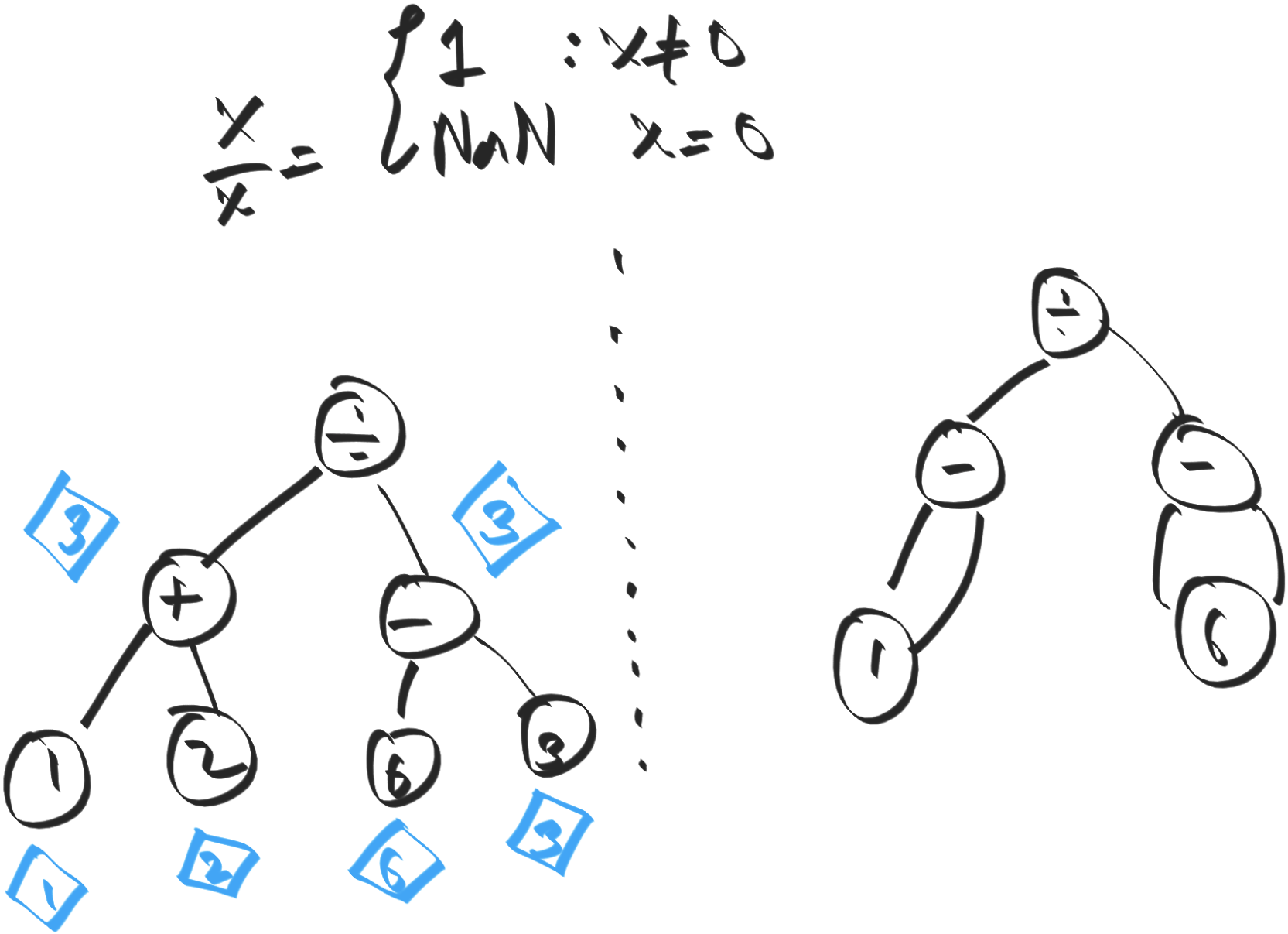
Herbie: Using egg — Analysis and rewrite



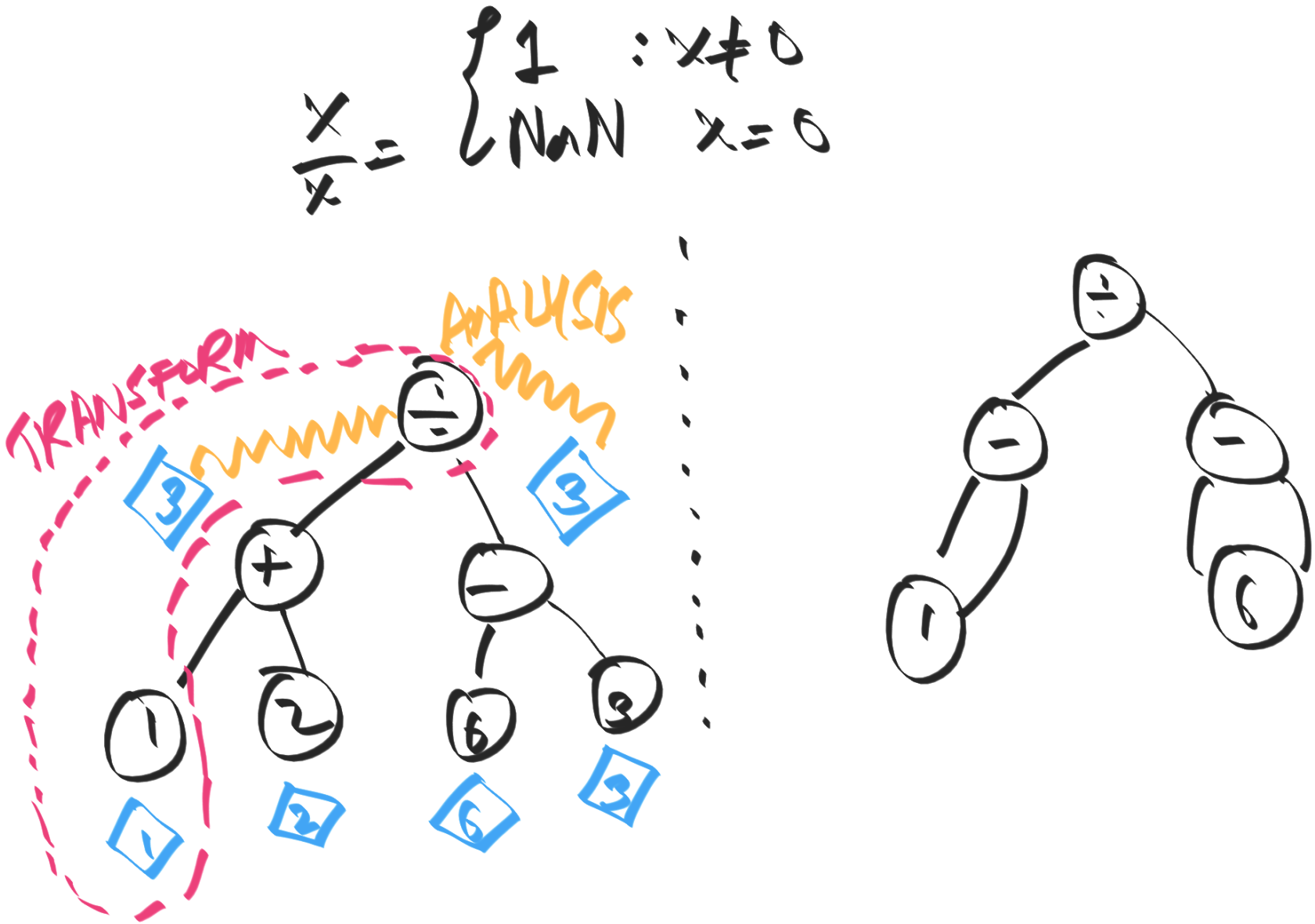
Herbie: Using egg — Analysis and rewrite



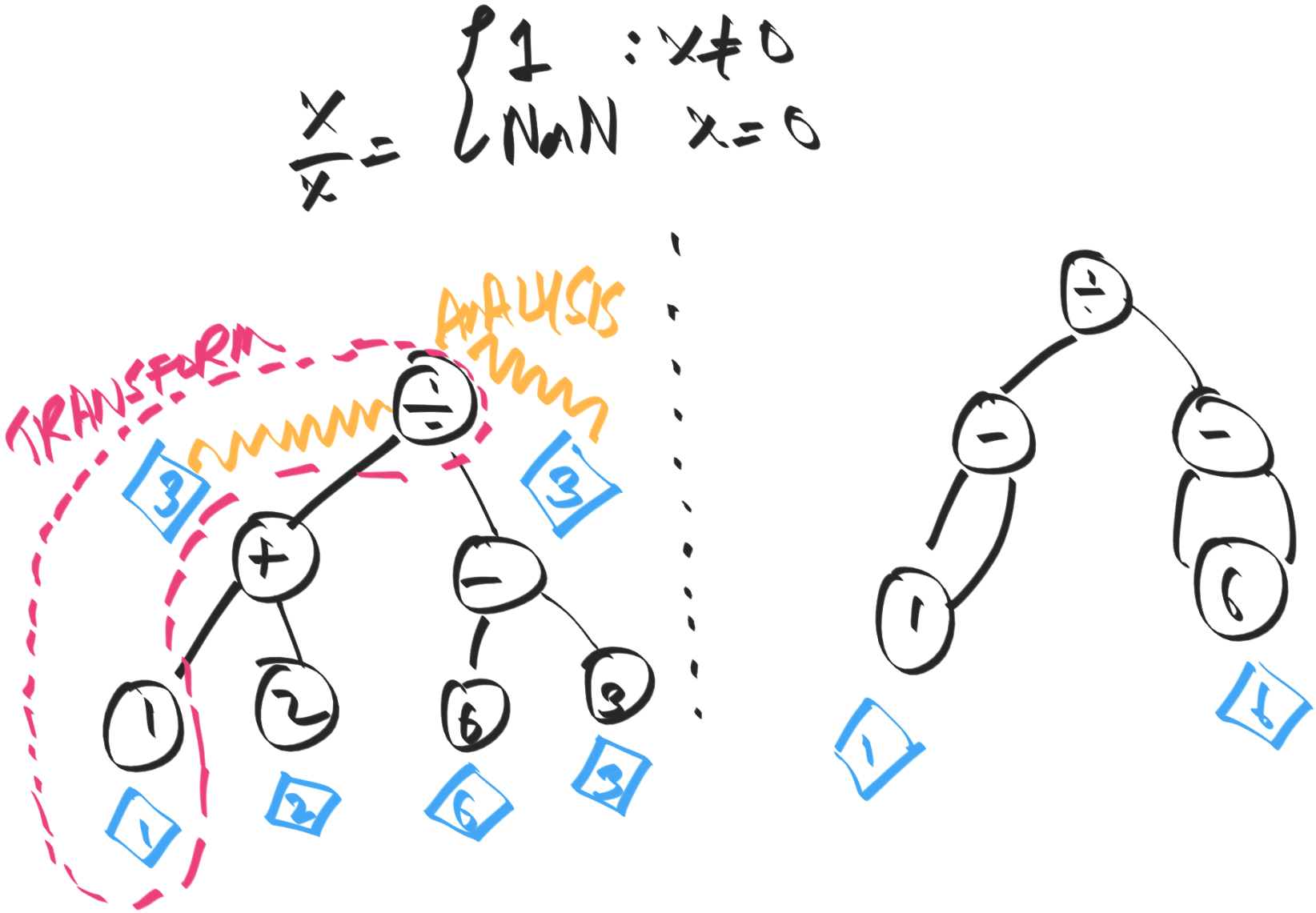
Herbie: Using egg — Analysis and rewrite



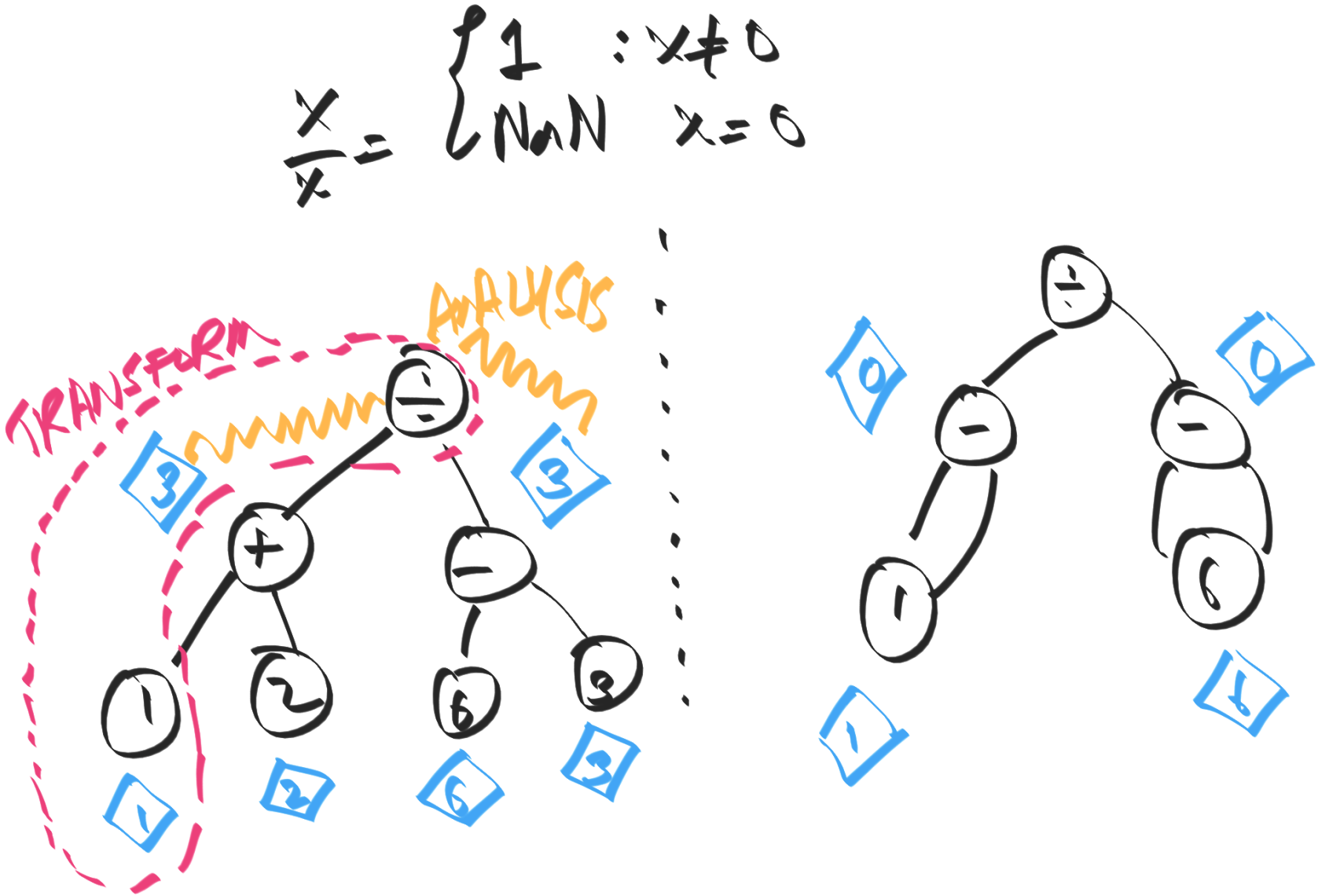
Herbie: Using egg — Analysis and rewrite



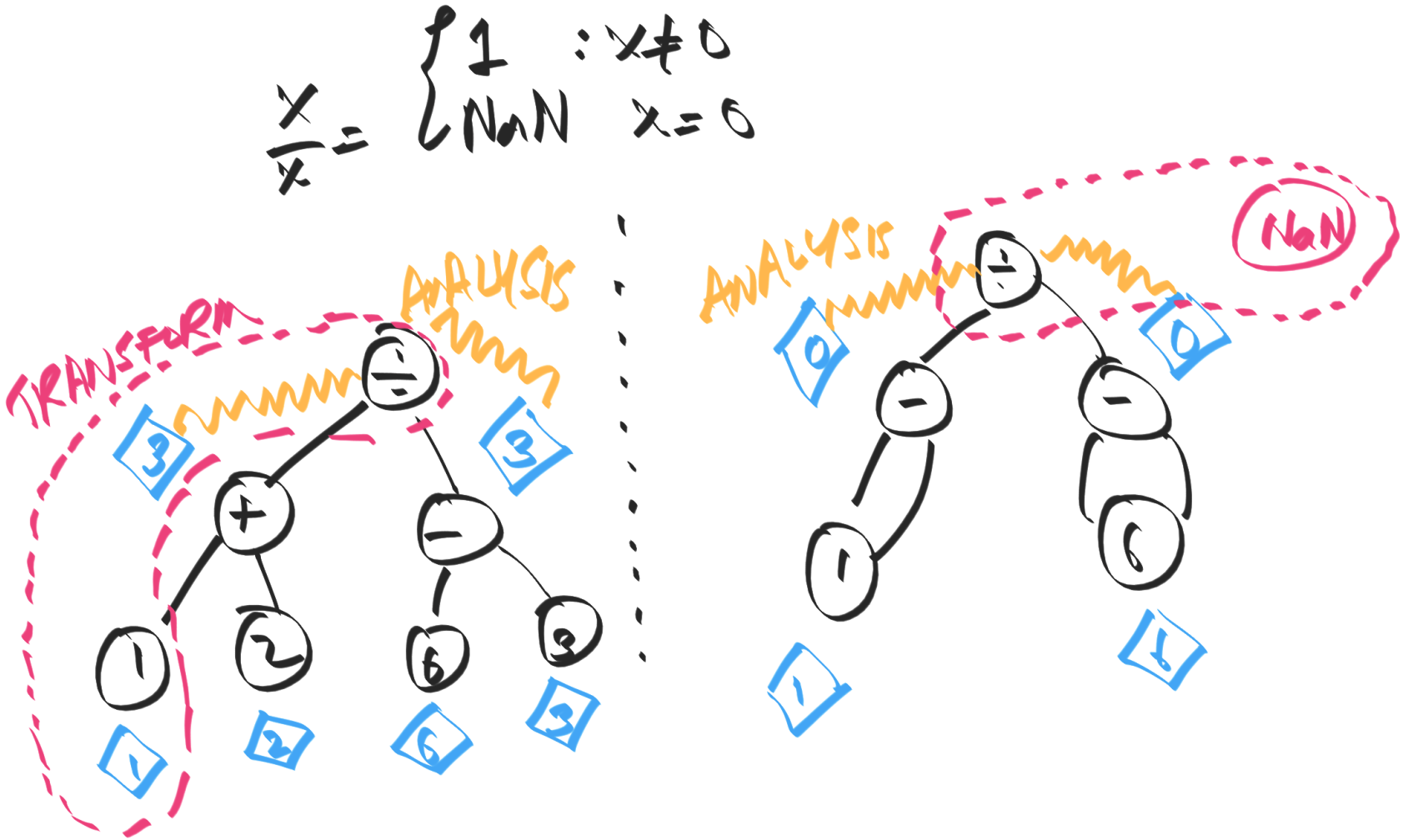
Herbie: Using egg — Analysis and rewrite



Herbie: Using egg — Analysis and rewrite



Herbie: Using egg — Analysis and rewrite



Lambda calculus in egg : Dynamic rewrites redux

```
-- v2=x is not free in (\x. x + 1)
x_bound = let v1 = (\x. x*2) in (\x.    x+1)
x_bound = let v1 = e         in (\v2.   body)
```

Lambda calculus in egg : Dynamic rewrites redux

```
-- v2=x is not free in (\x. x + 1)
x_bound = let v1 = (\x. x*2) in (\x.    x+1)
x_bound = let v1 = e         in (\v2.   body)
```

How to push `let` into `lambda`?

```
x_bound' = \x.  let v1 = (\x. x*2) in x + 1
x_bound' = \v2. let v1 = e         in body
```

Lambda calculus in egg : Dynamic rewrites redux

```
-- v2=x is not free in (\x. x + 1)
x_bound = let v1 = (\x. x*2) in (\x.    x+1)
x_bound = let v1 = e         in (\v2.   body)
```

How to push `let` into `lambda`?

```
x_bound' = \x.  let v1 = (\x. x*2) in x + 1
x_bound' = \v2. let v1 = e         in body
```

```
-- v2=x is free in e=x*2
x :: Int; x = 42
x_free = let v1 = x*2 in (\x.    x+1)
x_free = let v1 = e   in (\v2.   body)
```

Lambda calculus in egg : Dynamic rewrites redux

```
-- v2=x is not free in (\x. x + 1)
x_bound = let v1 = (\x. x*2) in (\x.    x+1)
x_bound = let v1 = e         in (\v2.   body)
```

How to push `let` into `lambda`?

```
x_bound' = \x.  let v1 = (\x. x*2) in x + 1
x_bound' = \v2. let v1 = e         in body
```

```
-- v2=x is free in e=x*2
x :: Int; x = 42
x_free = let v1 = x*2 in (\x.    x+1)
x_free = let v1 = e   in (\v2.   body)
```

```
-- v2=x is free in e=x*2
x :: Int; x = 42
x_free'_wrong = \x. let v1 = x*2 in x + 1
```

ERR!

Lambda calculus in egg : Dynamic rewrites redux

```
-- v2=x is not free in (\x. x + 1)
x_bound = let v1 = (\x. x*2) in (\x.    x+1)
x_bound = let v1 = e         in (\v2.   body)
```

How to push `let` into `lambda`?

```
x_bound' = \x.  let v1 = (\x. x*2) in x + 1
x_bound' = \v2. let v1 = e         in body
```

```
-- v2=x is free in e=x*2
x :: Int; x = 42
x_free = let v1 = x*2 in (\x.    x+1)
x_free = let v1 = e   in (\v2.   body)
```

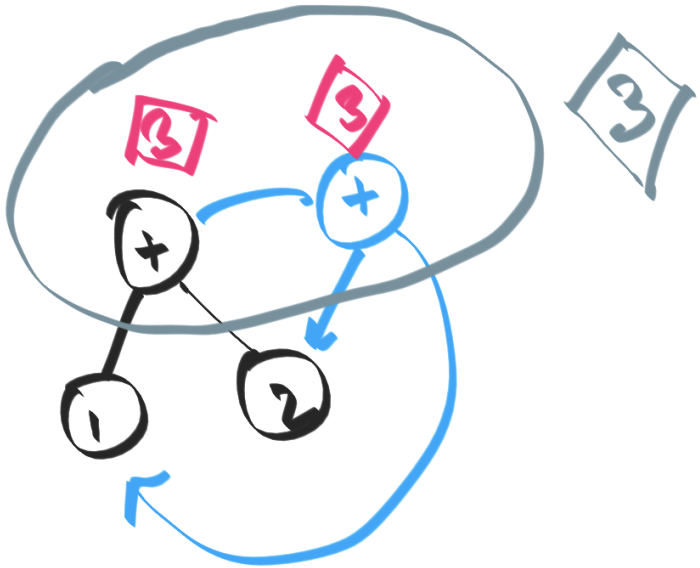
```
-- v2=x is free in e=x*2
x :: Int; x = 42
x_free'_wrong = \x. let v1 = x*2 in x + 1
```

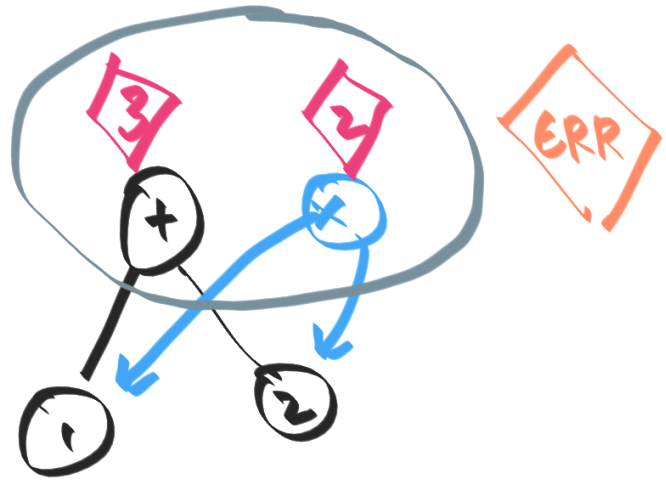
ERR!

How to push `let` into `lambda`?

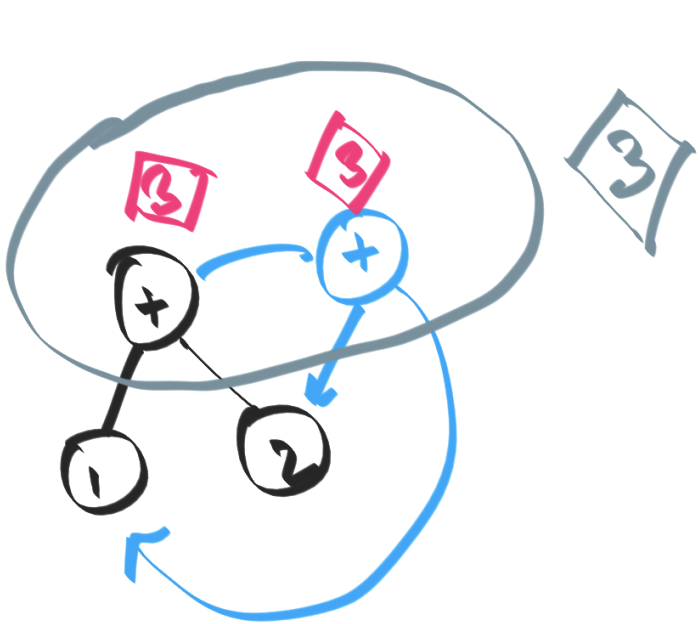
```
x :: Int; x = 42
x_free' = \fresh. let v1 = e   in (let v2 = fresh in body  )
x_free' = \fresh. let v1 = x*2 in (let x  = fresh in (x + 1))
```

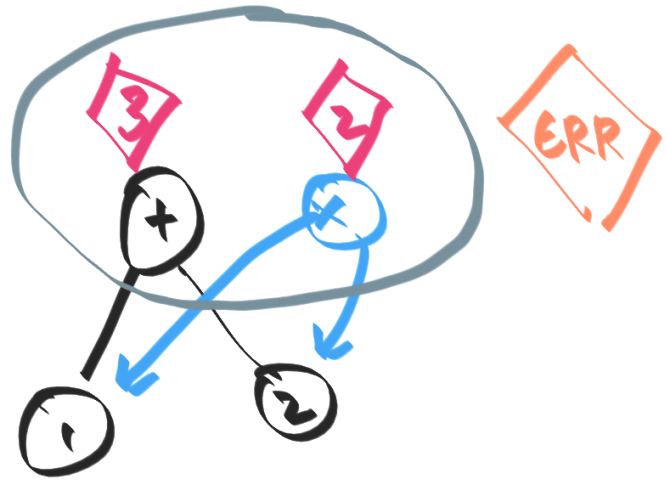
Herbie: Using egg — Lattices





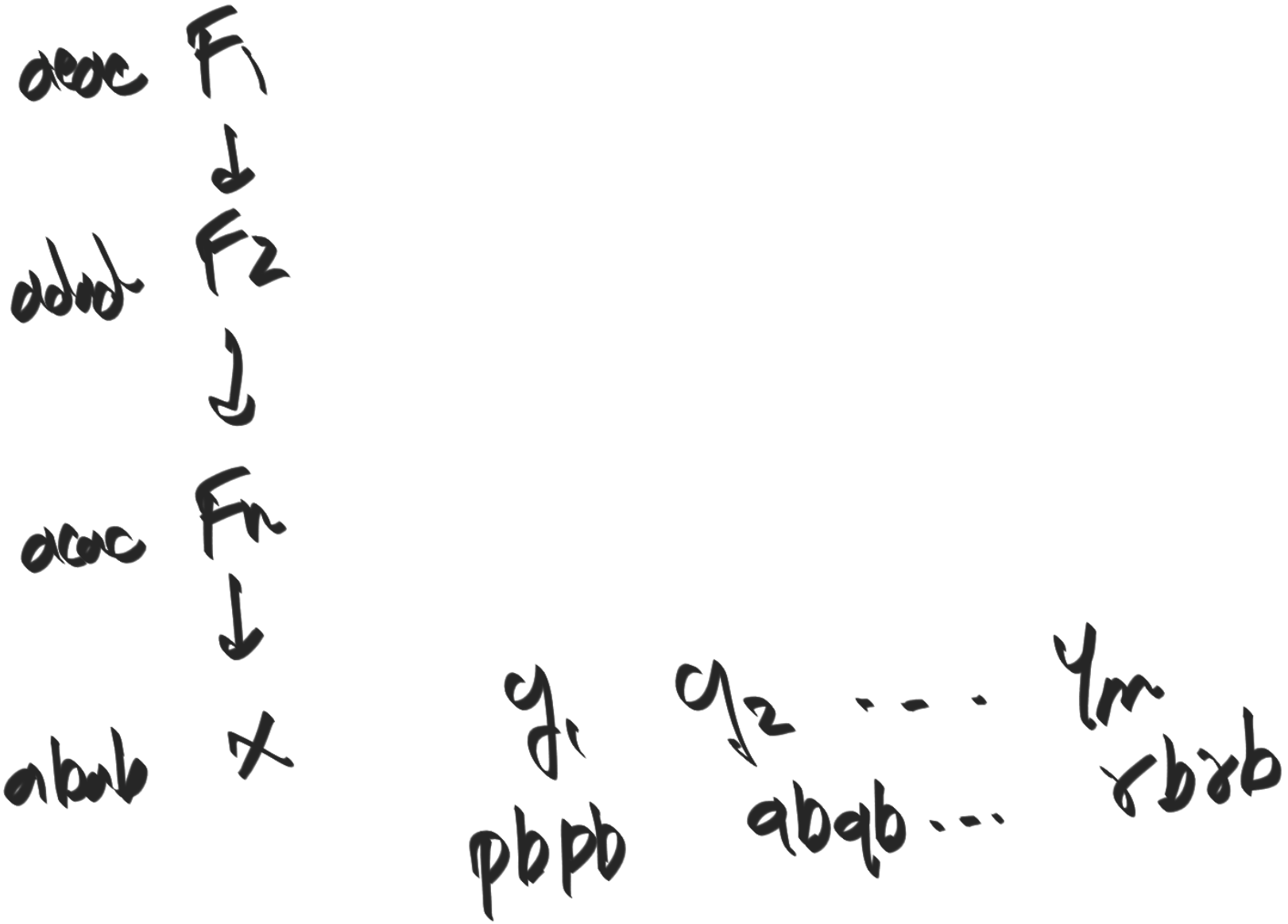
Herbie: Using egg — Lattices



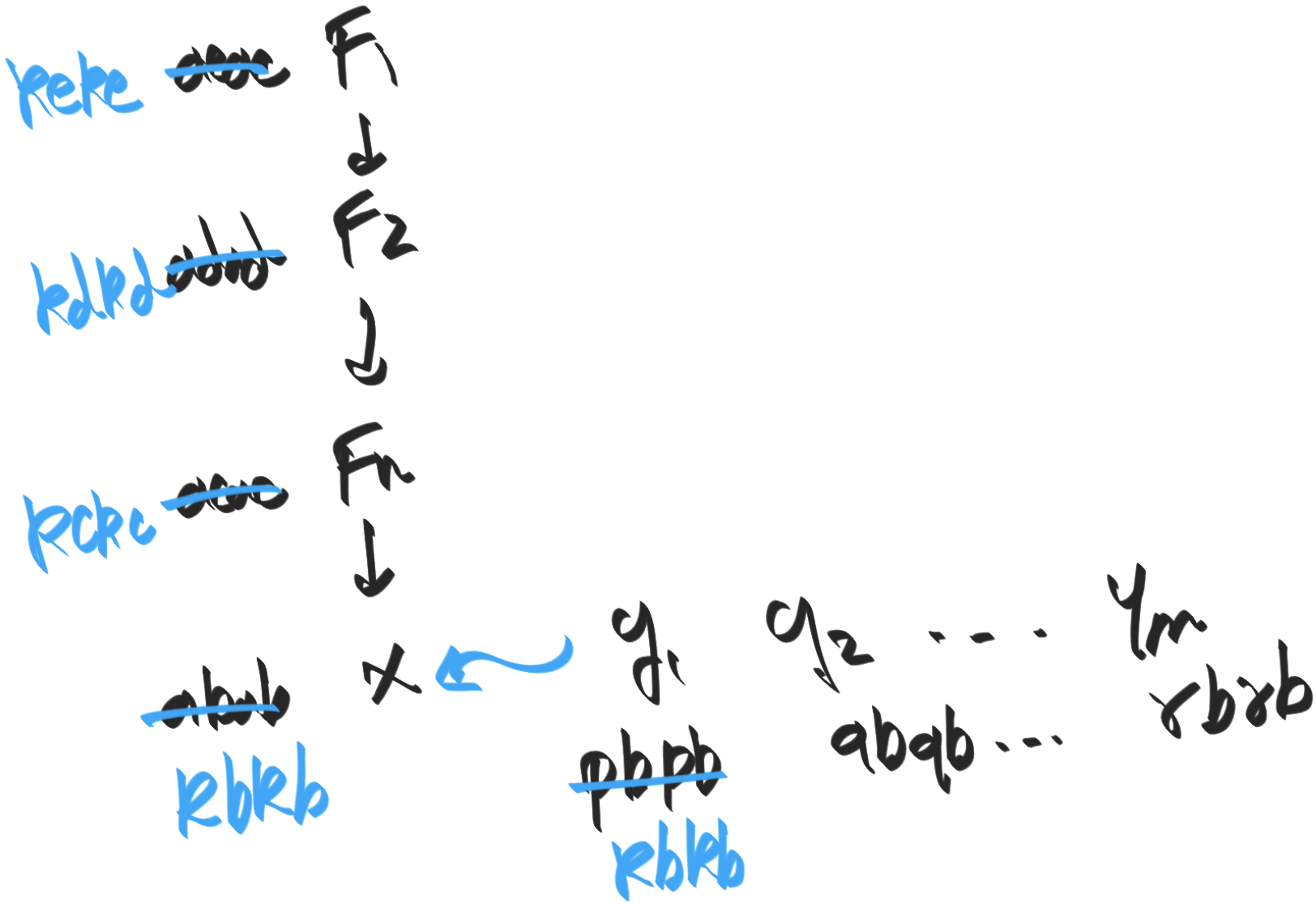


- Abstract interpretation of equivalence classes.
- For each node, provide function $\alpha : \texttt{node} \to L$ (Abstraction function)
- $(L, \cap)$ is a join-semilattice.
- egg provides for each equivalence class $\texttt{class} \mapsto \bigcap_{\texttt{node} \in \texttt{class}} \alpha(\texttt{node}) \in L$

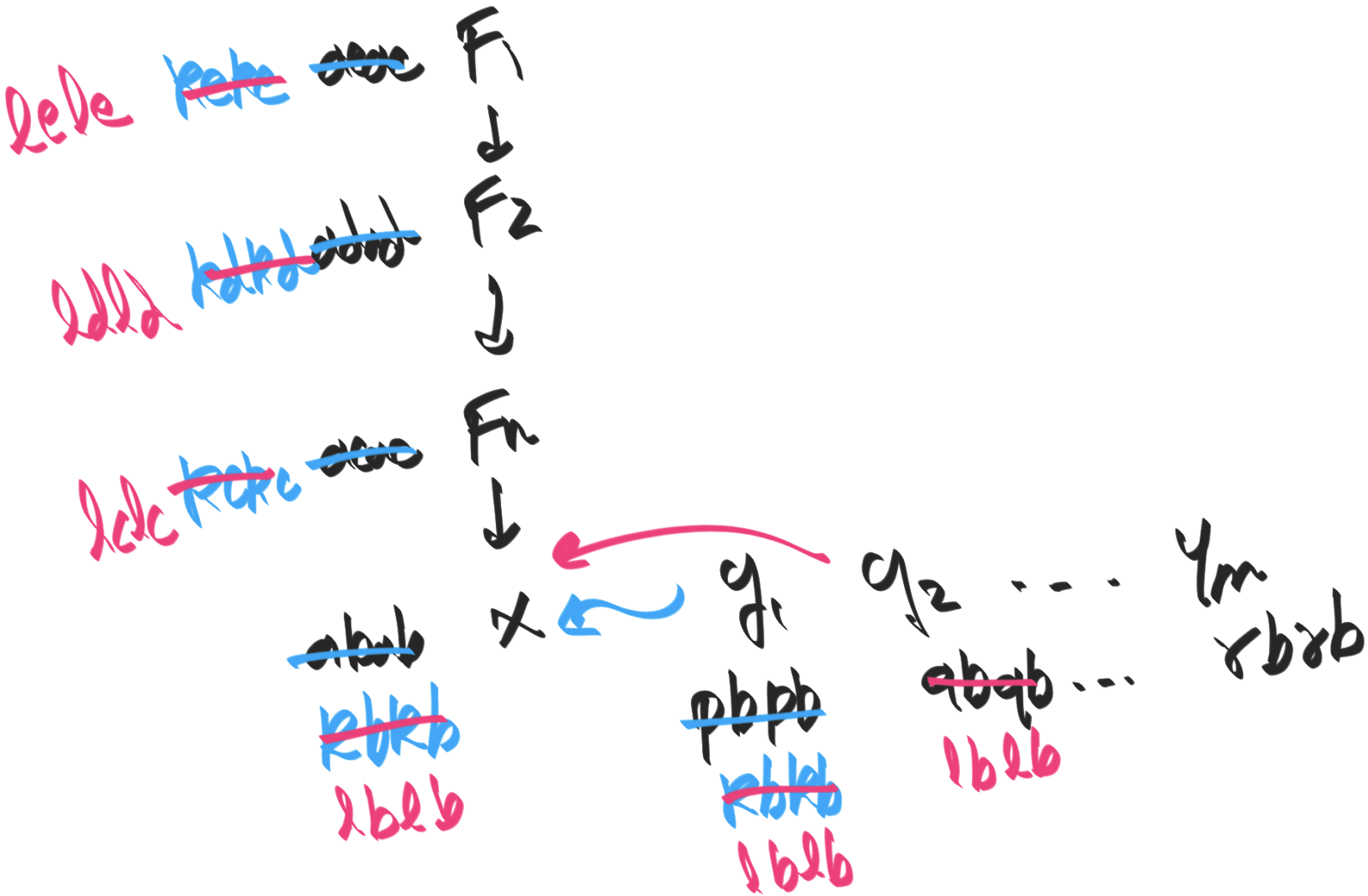
Speedup over prior art: Merging



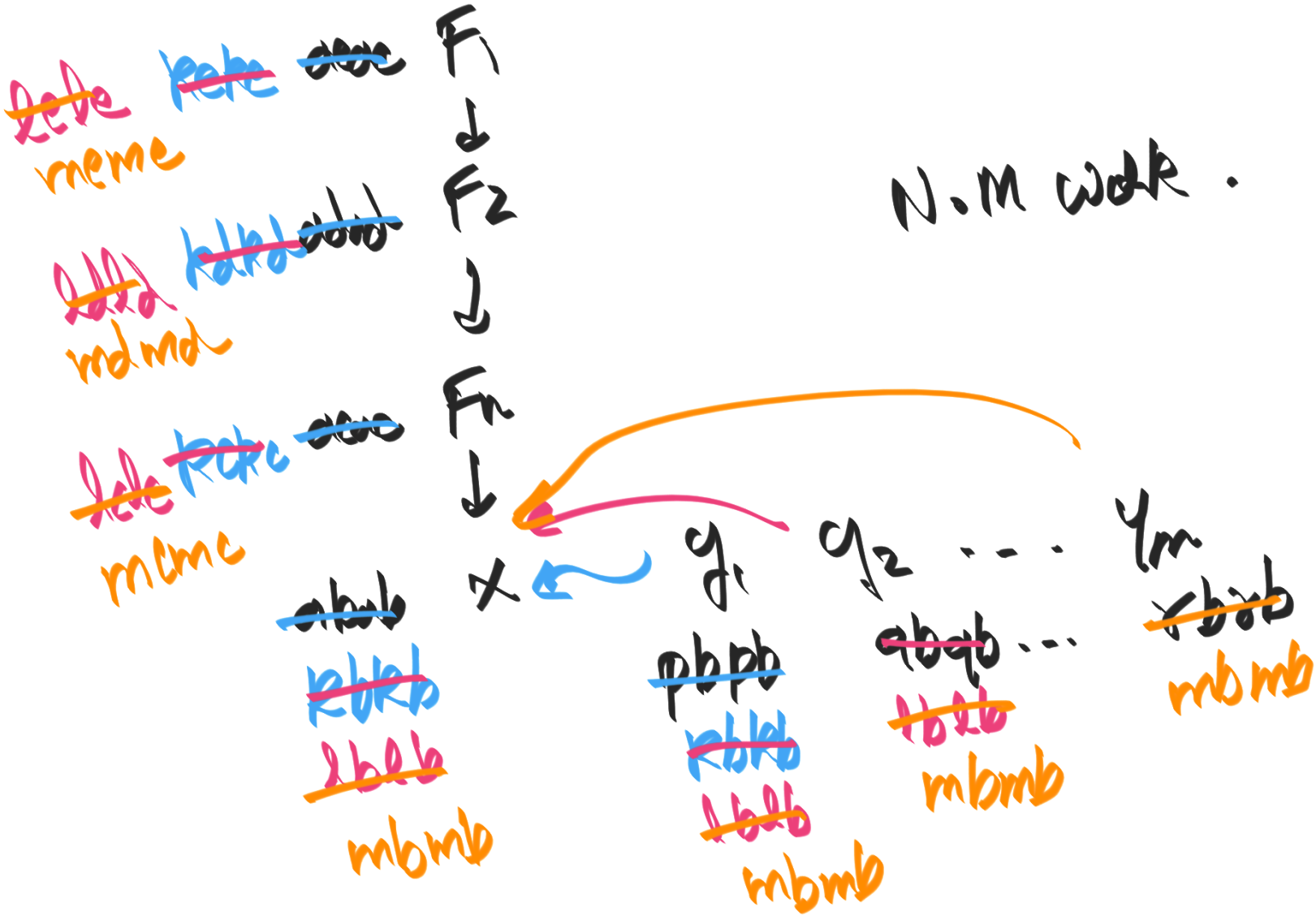
Speedup over prior art: Merging



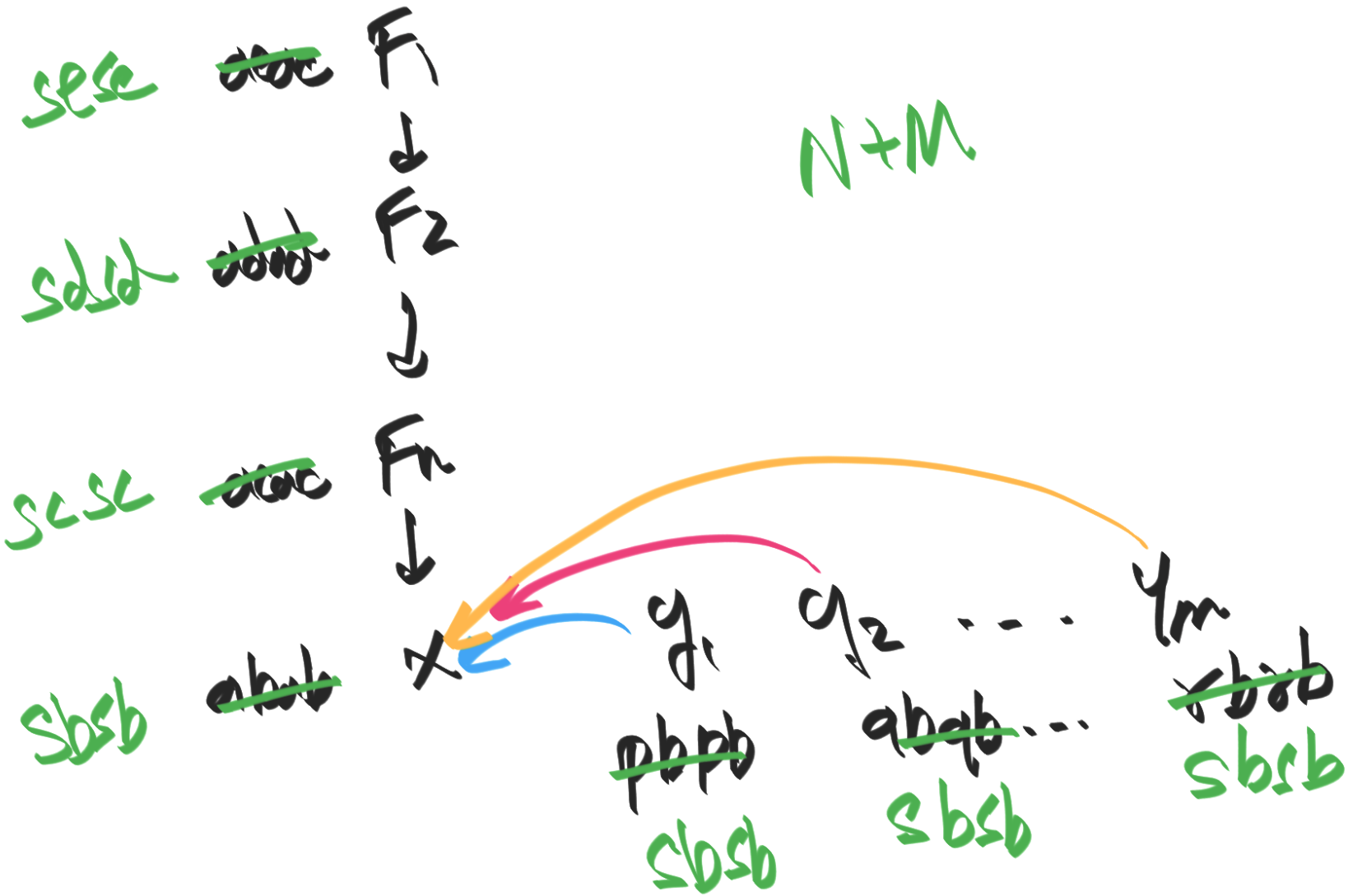
Speedup over prior art: Merging



Speedup over prior art: Merging



Speedup over prior art: Merging



Speedup over prior art: Merging

3.2.1 Examples of Rebuilding. Deferred rebuilding speeds up congruence maintenance by amortizing the work of maintaining the hashcons invariant. Consider the following terms in an e-graph: $f_1(x), ..., f_n(x), y_1, ..., y_n$. Let the workload be $\texttt{merge}(x, y_1), ..., \texttt{merge}(x, y_n)$. Each merge may change the canonical representation of the $f_i(x)$s, so the traditional invariant maintenance strategy could require $O(n^2)$ hashcons updates. With deferred rebuilding the `merges` happen before the hashcons invariant is restored, requiring no more than $O(n)$ hashcons updates.