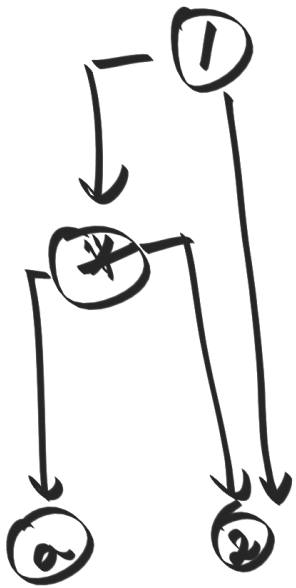
egg : Fast and extensible equality saturation

Siddharth Bhat

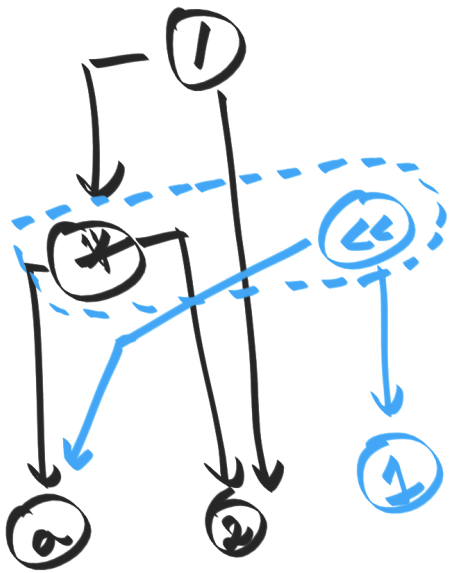
Monday, Jan 18 2021

egg : Fast and extensible equality saturation



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

egg : Fast and extensible equality saturation (2)

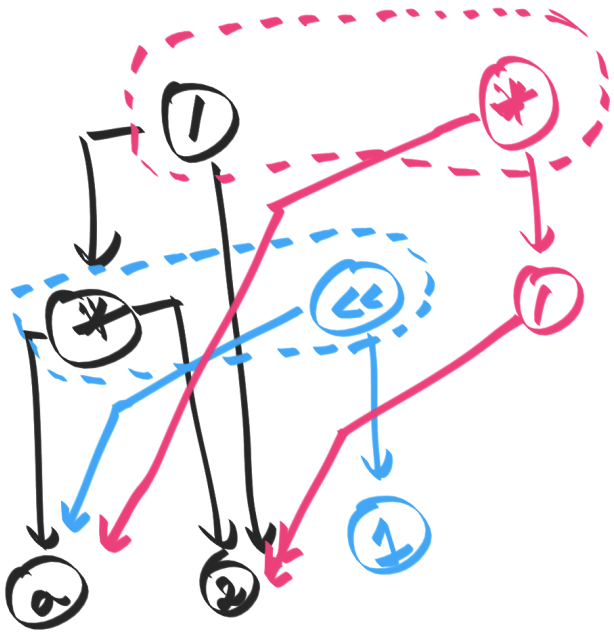


(a*2)/2:
(/ (* a 2) 2)

p*2 → p<<1

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

egg : Fast and extensible equality saturation (3)



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

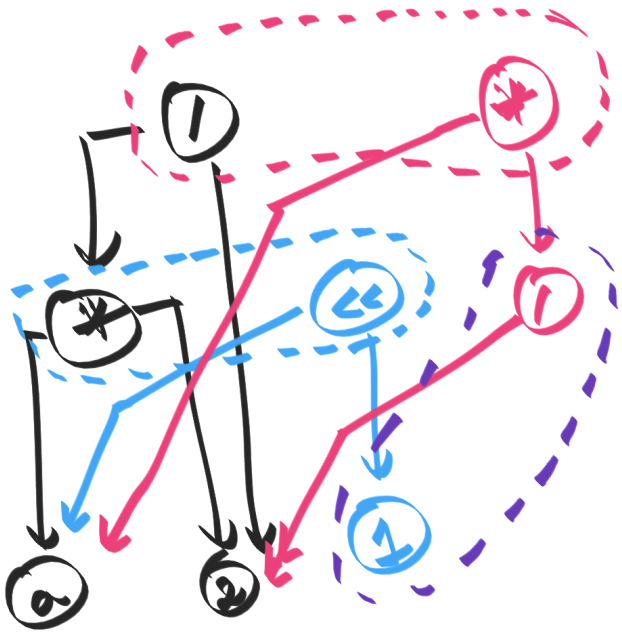
p*2 → p<<1

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

(p*a)/2 → p*(a/2)

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

egg : Fast and extensible equality saturation (4)



$x/x \to 1$

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

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

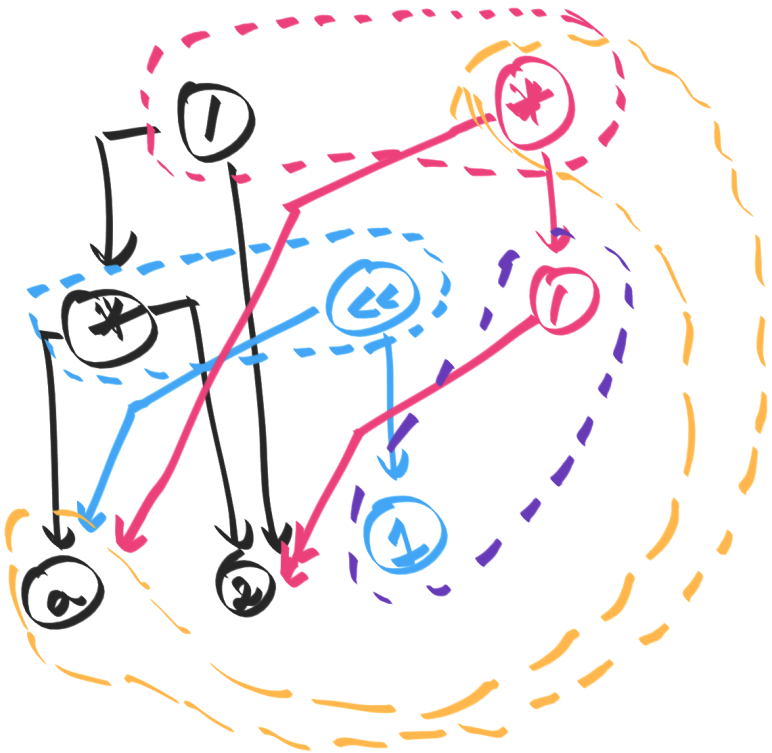
$p*2 \to p<<1$

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

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

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

egg : Fast and extensible equality saturation (5)



$x/x \to 1$

$(* a (/ 2 2))$

$x * 1 \to x$

$(* a (/ 2 2))$

a

$(a*2)/2:$

$(/ (* a 2) 2)$

$p*2 \to p << 1$

$(/ (* a 2) 2)$

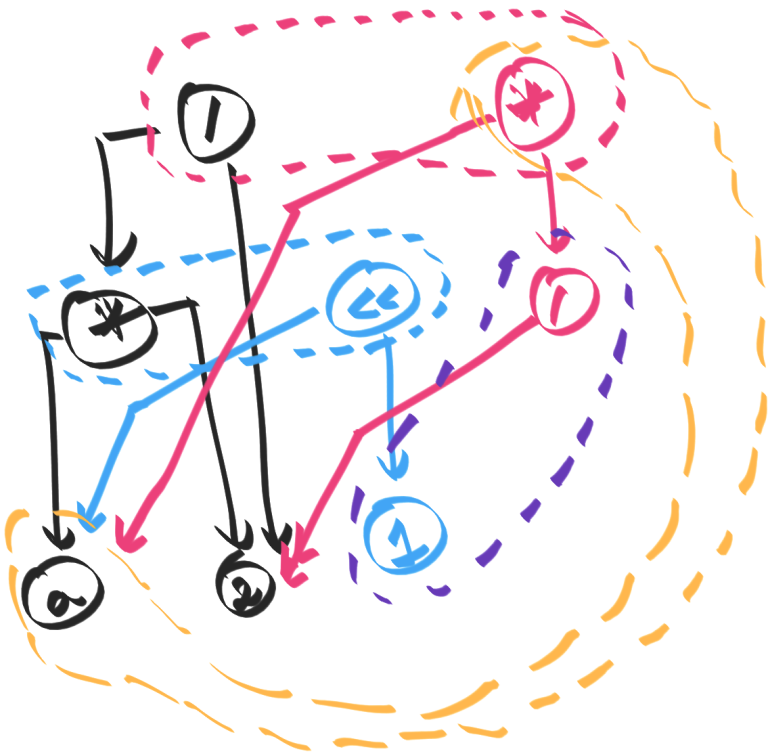
$(<< a 1)$

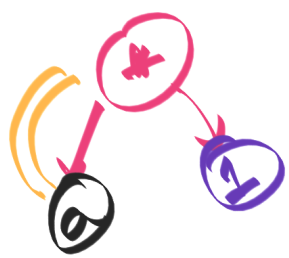
$(p*a)/2 \to p*(a/2)$

$(/ (* a 2) 2)$

$(* a (/ 2 2))$

egg : Fast and extensible equality saturation (6)





$x/x \to 1$

$(* a (/ 2 2))$

a 1

$x*1 \to x$

$(* a (/ 2 2))$

a 1

$(a*2)/2$:

$(/ (* a 2) 2)$

$p*2 \to p << 1$

$(/ (* a 2) 2)$

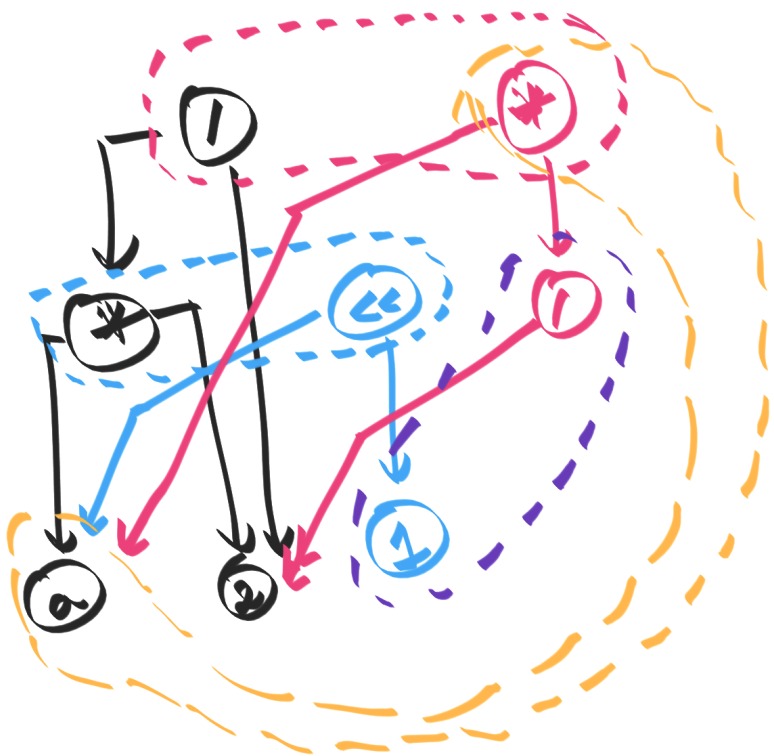
$(<< a 1)$

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

$(/ (* a 2) 2)$

$(* a (/ 2 2))$

egg : Fast and extensible equality saturation (7)



$x/x \to 1$

$(* a (/ 2 2))$ → $(* a 1)$

$x * 1 \to x$

$(* a (/ 2 2))$ → $(* a 1)$ → a

$(a*2)/2$:

$(/ (* a 2) 2)$

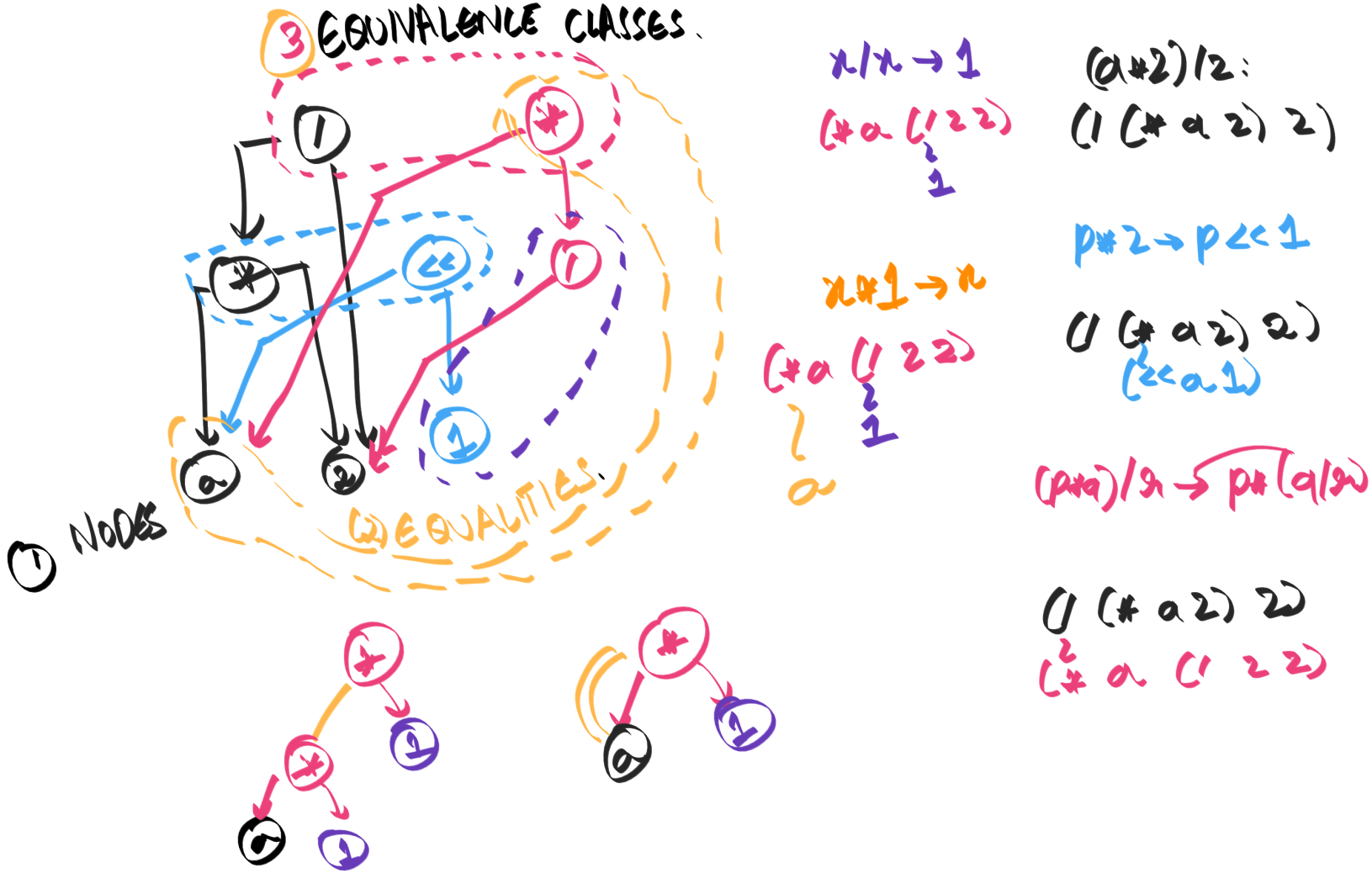
$p * 2 \to p << 1$

$(/ (* a 2) 2)$ → $(<< a 1)$

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

$(/ (* a 2) 2)$ → $(* a (/ 2 2))$

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.

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

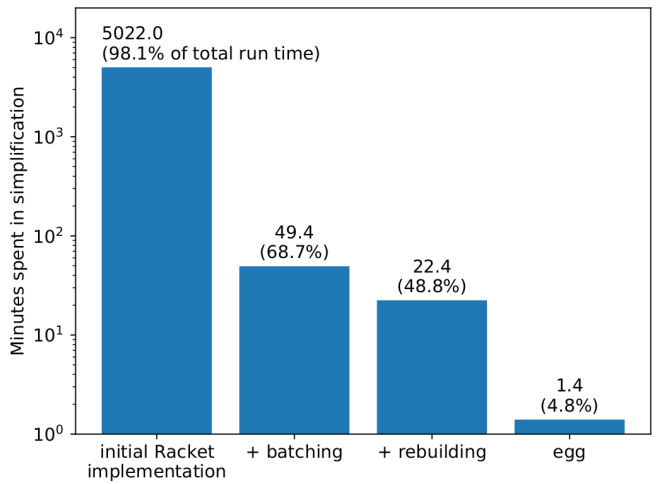


Fig. 12. Herbie sped up its expression simplification phase by adopting egg-inspired features like batched simplification and rebuilding into its Racket-based e-graph implementation. Herbie also supports using egg itself for additional speedup. Note that the y-axis is log-scale.

Herbie: Using egg — rules

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/rules.rs

```
;; projections
("real-part","(re real (complex real ?x ?y))","?x")
;; why duplication of real and complex rules?
("associate-+r+.c", "(+ c ?a (+ c ?b ?c))", "(+ c (+ c ?a ?b) ?c)")
("associate-+r+", "(+ f64 ?a (+ f64 ?b ?c))", "(+ f64 (+ f64 ?a ?b) ?c)")
;; branch simplification
("if-true", "(if real (TRUE real) ?x ?y)", "?x")
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;

impl Analysis<Math> for ConstantFold {
  type Data = Option<Constant>; // data for each equiv-class
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;

impl Analysis<Math> for ConstantFold {
  type Data = Option<Constant>; // data for each equiv-class

  fn make(egraph: &EGraph, enode: &Math) -> Self::Data {
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;

impl Analysis<Math> for ConstantFold {
  type Data = Option<Constant>; // data for each equiv-class

  fn make(egraph: &EGraph, enode: &Math) -> Self::Data {

    let x = |id: &Id| egraph[*id].data.as_ref(); // access Data
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;

impl Analysis<Math> for ConstantFold {
  type Data = Option<Constant>; // data for each equiv-class

  fn make(egraph: &EGraph, enode: &Math) -> Self::Data {

    let x = |id: &Id| egraph[*id].data.as_ref(); // access Data

    match enode {
     Math::Add([_p, a, b]) => Some(x(a)? + x(b)?),
```

Herbie: Using egg — analysis

https://github.com/uwplse/herbie/blob/master/egg-herbie/src/math.rs

```
define_language! {
    pub enum Math {
        "+" = Add([Id; 3]),
        "-" = Sub([Id; 3]),
        ...
    }
}

pub struct ConstantFold { ... }

pub type Constant = num_rational::BigRational;

impl Analysis<Math> for ConstantFold {
  type Data = Option<Constant>; // data for each equiv-class

  fn make(egraph: &EGraph, enode: &Math) -> Self::Data {

    let x = |id: &Id| egraph[*id].data.as_ref(); // access Data

    match enode {
     Math::Add([_p, a, b]) => Some(x(a)? + x(b)?),

      Math::Sqrt([_p, a]) => {
        let a = x(a)?;
        if *a.numer() > BigInt::from(0) && *a.denom() > BigInt::from(0) {
            let s1 = a.numer().sqrt();
            let s2 = a.denom().sqrt();
            let is_perfect = &(&s1 * &s1) == a.numer()
                && &(&s2 * &s2) == a.denom();
            if is_perfect { Some(Ratio::new(s1, s2)) } else { None }
        } else { None }
    }
  }
}
```

Herbie: Using egg — Lattice

Herbie: Using egg — Lattice

```
// commutative, associative, idempotent, has identity element (Nothing)
fn merge(&self, to: &mut Self::Data, from: Self::Data) -> bool {
```

Herbie: Using egg — Lattice

```
// commutative, associative, idempotent, has identity element (Nothing)
fn merge(&self, to: &mut Self::Data, from: Self::Data) -> bool {

    match (&to, from) {
        (None, None) => false,
```

Herbie: Using egg — Lattice

```
// commutative, associative, idempotent, has identity element (Nothing)
fn merge(&self, to: &mut Self::Data, from: Self::Data) -> bool {

    match (&to, from) {
        (None, None) => false,

        (Some(_), None) => false, // no update needed
```

Herbie: Using egg — Lattice

```
// commutative, associative, idempotent, has identity element (Nothing)
fn merge(&self, to: &mut Self::Data, from: Self::Data) -> bool {
    match (&to, from) {
        (None, None) => false,
        (Some(_), None) => false, // no update needed
        (None, Some(c)) => {
            *to = Some(c);
            true
        }
```

Herbie: Using egg — Lattice

```rust
// commutative, associative, idempotent, has identity element (Nothing)
fn merge(&self, to: &mut Self::Data, from: Self::Data) -> bool {
    match (&to, from) {
        (None, None) => false,
        (Some(_), None) => false, // no update needed
        (None, Some(c)) => {
            *to = Some(c);
            true
        }
        (Some(a), Some(ref b)) => {
            // bad analysis detected!
            if a != b { log::warn!("Bad merge detected: {} != {}", a, b); }
            false
        }
    }
}
```

Herbie: Using egg — Extraction

```rust
pub struct Extracted { pub best: RecExpr, pub cost: usize, }
```

Herbie: Using egg — Extraction

```
pub struct Extracted { pub best: RecExpr, pub cost: usize, }

pub struct IterData { pub extracted: Vec<(Id, Extracted)>, }
impl IterationData<Math, ConstantFold> for IterData {
```

Herbie: Using egg — Extraction

```rust
pub struct Extracted { pub best: RecExpr, pub cost: usize, }

pub struct IterData { pub extracted: Vec<(Id, Extracted)>, }
impl IterationData<Math, ConstantFold> for IterData {

    fn make(runner: &Runner) -> Self {
        let mut extractor = Extractor::new(&runner.egraph, AstSize);
```

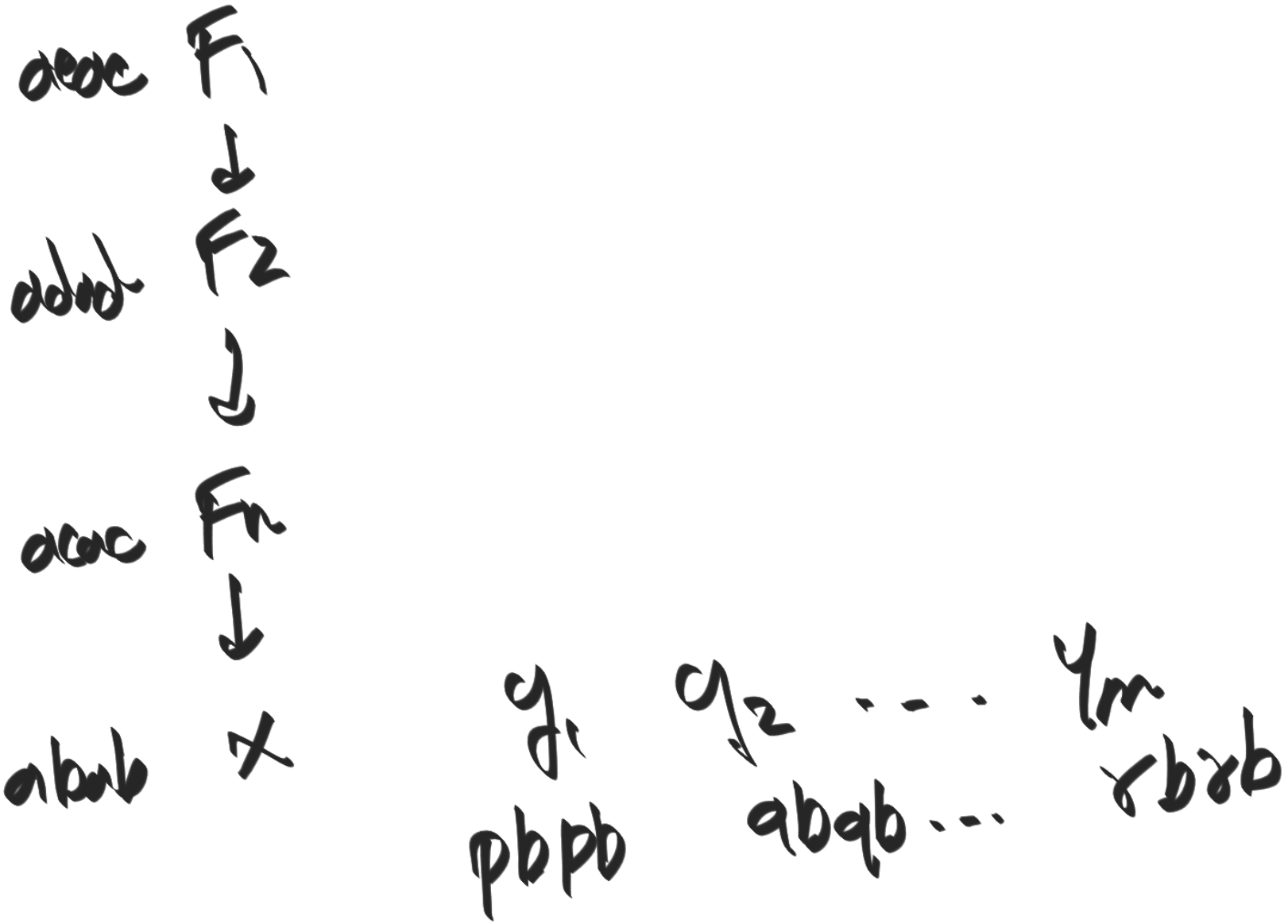
Herbie: Using egg — Extraction

```rust
pub struct Extracted { pub best: RecExpr, pub cost: usize, }

pub struct IterData { pub extracted: Vec<(Id, Extracted)>, }
impl IterationData<Math, ConstantFold> for IterData {

    fn make(runner: &Runner) -> Self {
        let mut extractor = Extractor::new(&runner.egraph, AstSize);

        let extracted = runner
            .roots
            .iter()
            .map(|&root| {
                let (cost, best) = extractor.find_best(root);
                let ext = Extracted { cost, best };
                (root, ext)
            })
            .collect();
        Self { extracted }
    }
}
```

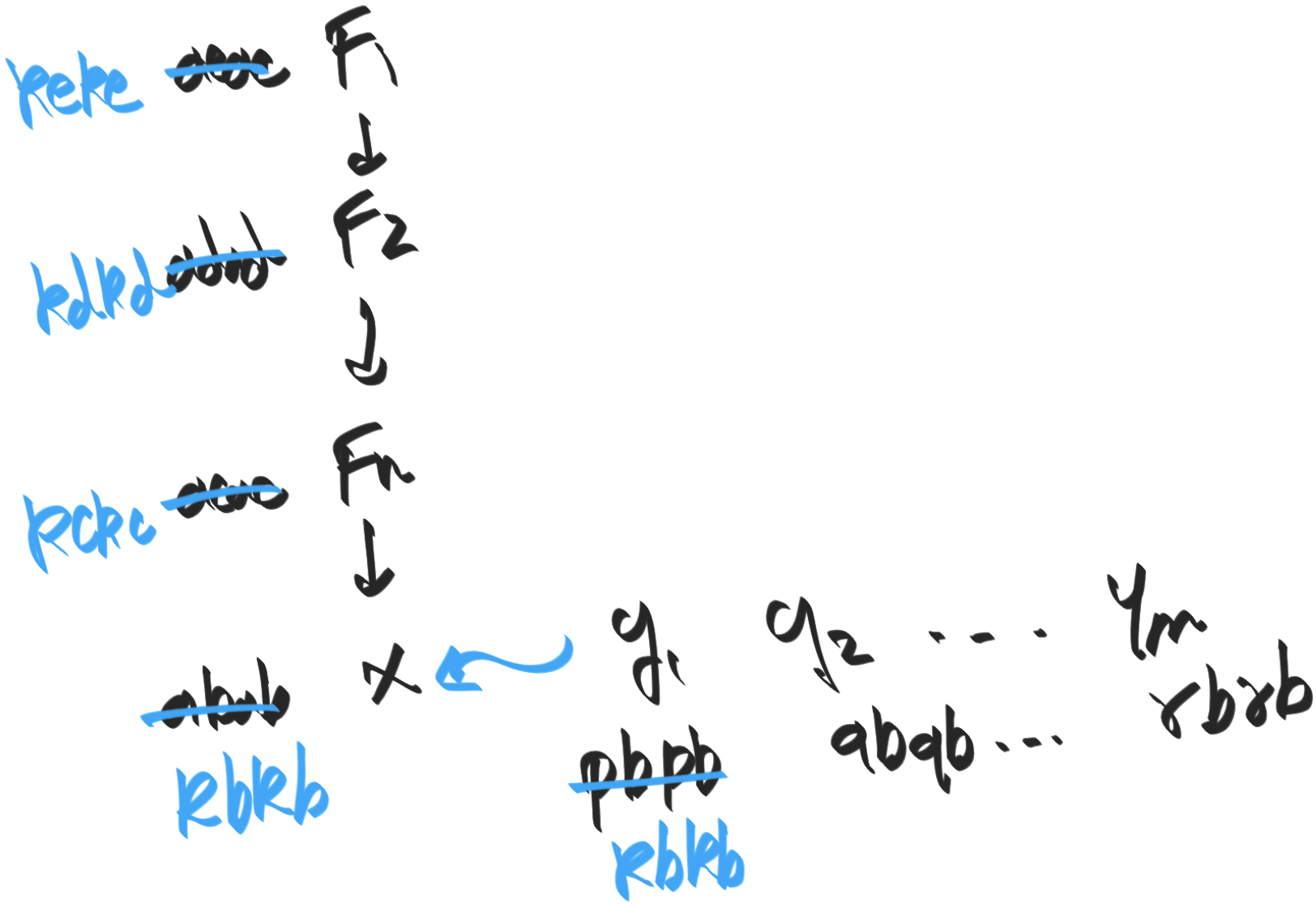
Architecture

- Read Phase: Find all matching rewrites
- Write Phase: Perform rewrites
- Canonicalize Phase: Merge equivalent nodes.

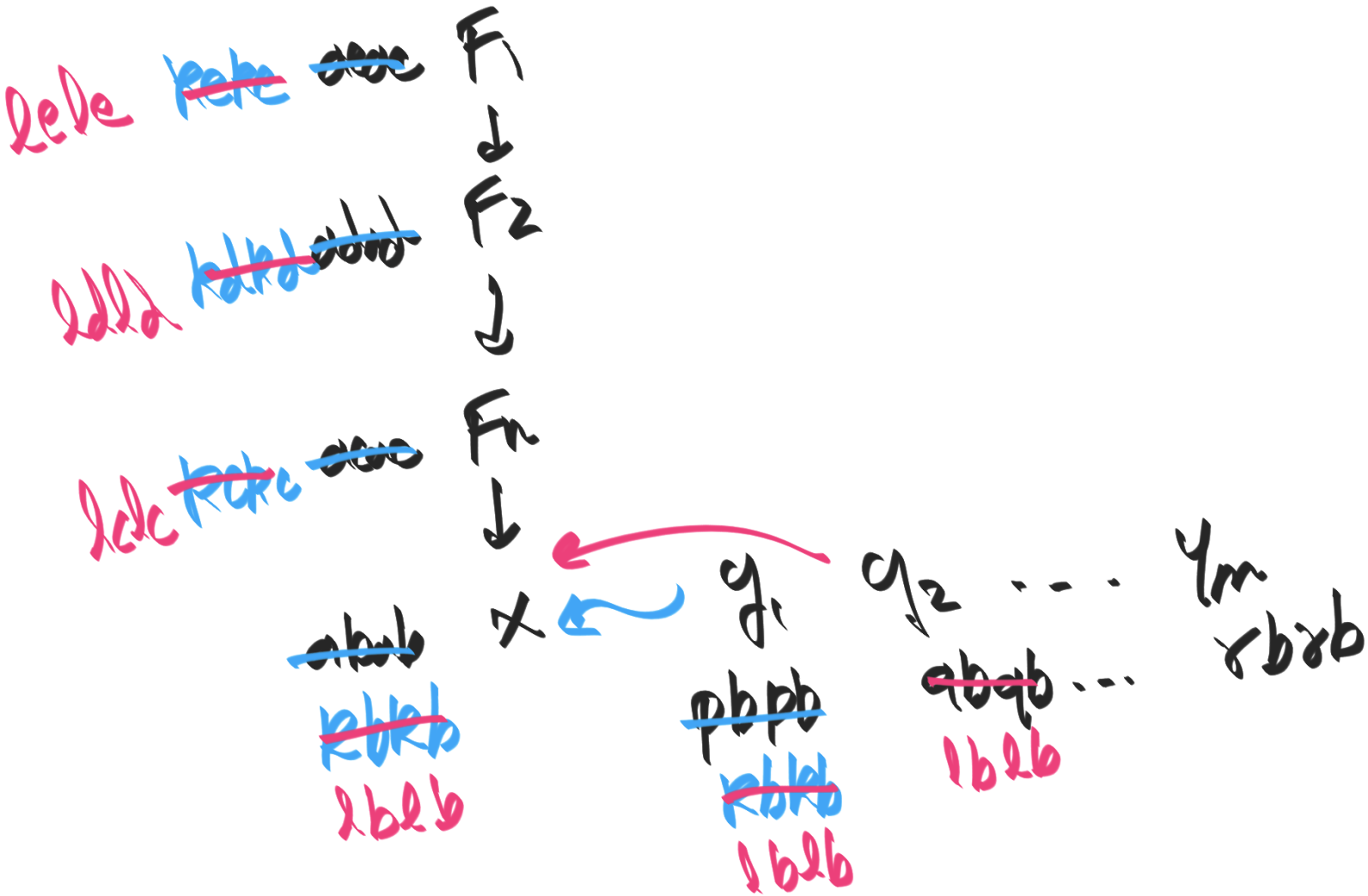
Speedup over prior art: Merging



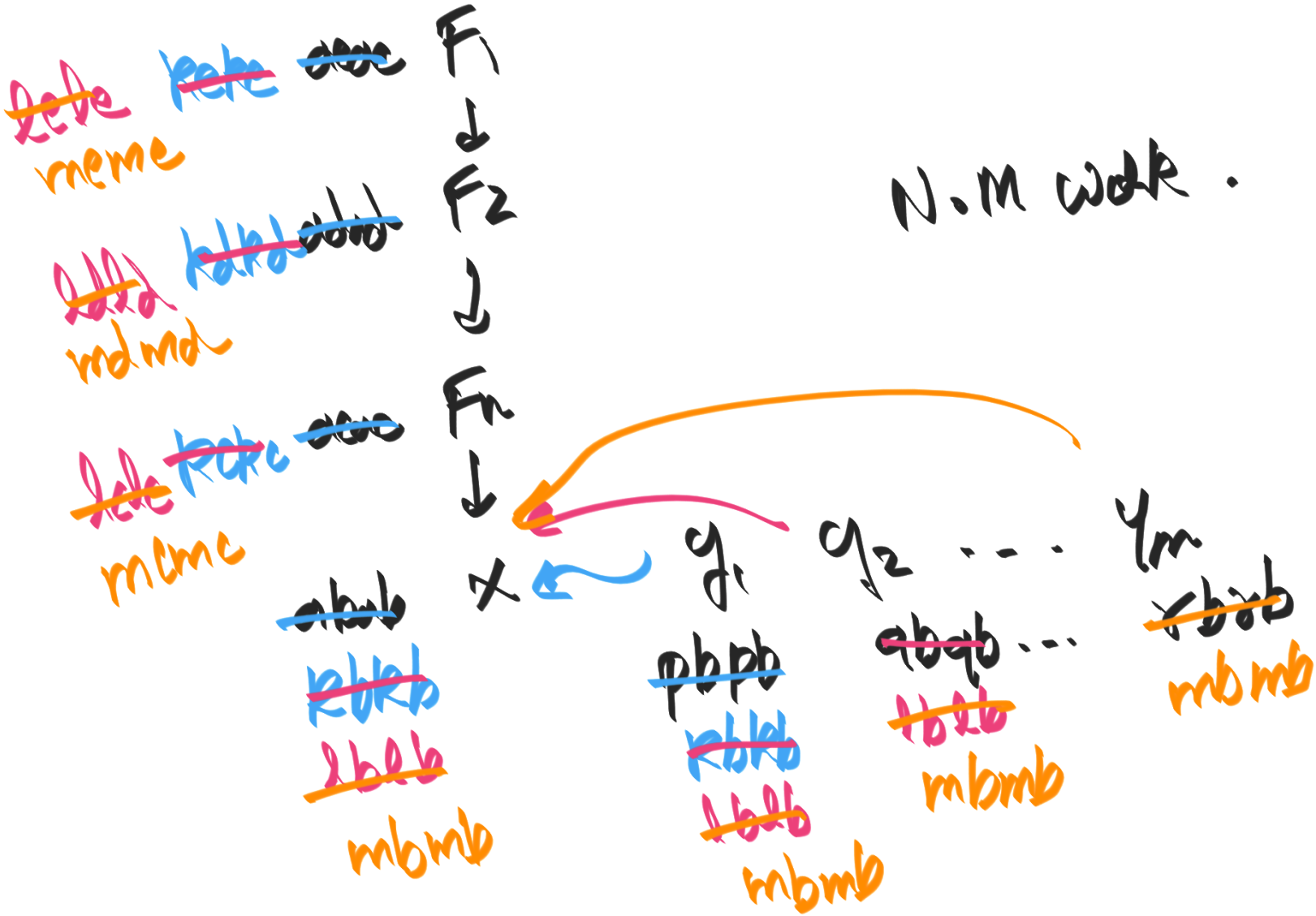
Speedup over prior art: Merging



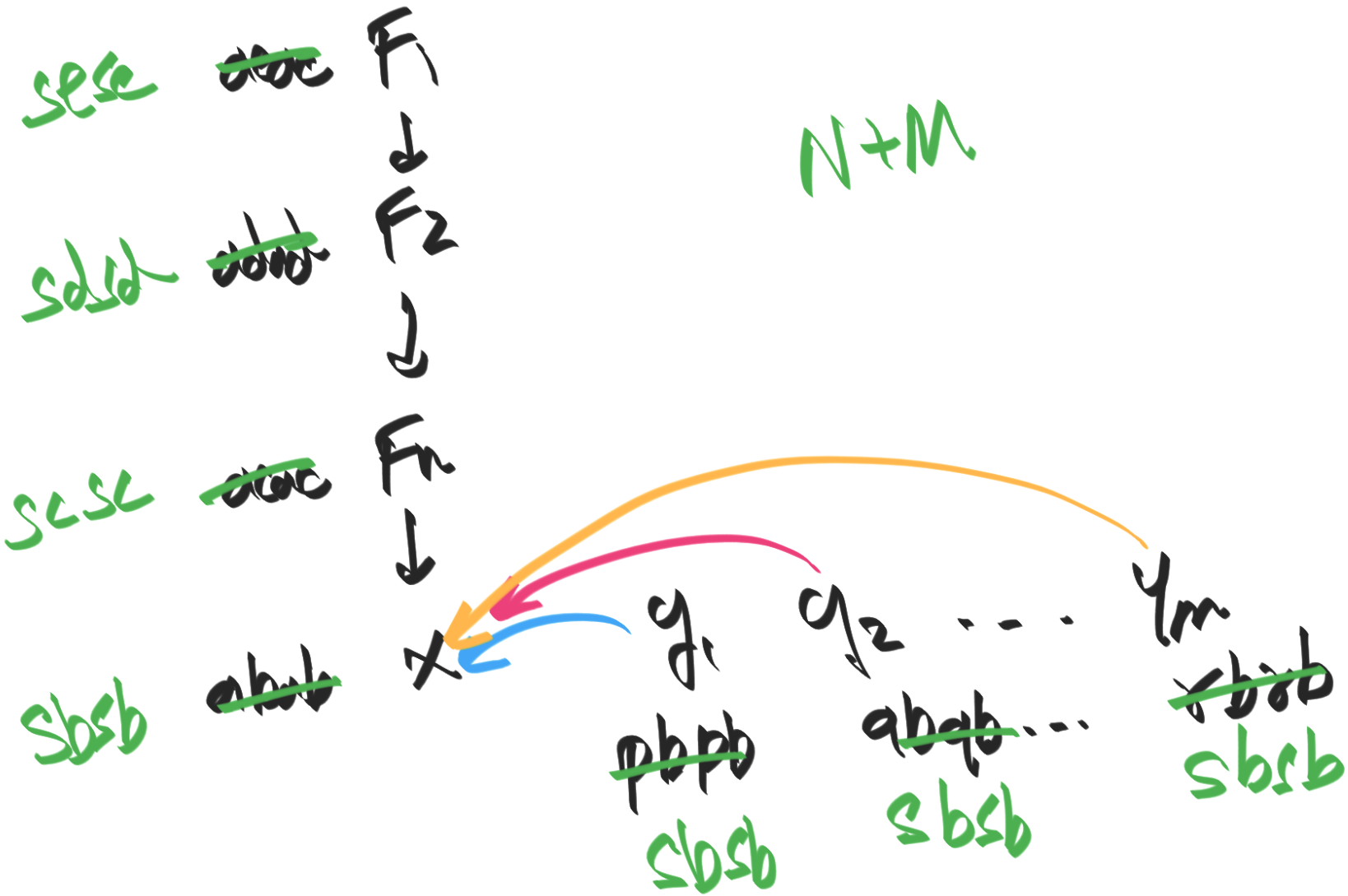
Speedup over prior art: Merging



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.

Analysis, formally

- 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 $\texttt{class} \mapsto \bigcap_{\texttt{node} \in \texttt{class}} \alpha(\texttt{node}) \in L$

Lambda calculus in egg : Language

```
define_language! {
  enum Lambda {
      Bool(bool),
      Num(i32),

      "var" = Var(Id),

      "+" = Add([Id; 2]),
      "=" = Eq([Id; 2]),

      "app" = App([Id; 2]),
      "lam" = Lambda([Id; 2]),
      "let" = Let([Id; 3]),
      "fix" = Fix([Id; 2]),

      "if" = If([Id; 3]),

      Symbol(egg::Symbol),
  }
}
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    fn make(egraph: &EGraph, enode: &Lambda) -> Data {
        let f = |i: &Id| egraph[*i].data.free.iter().cloned();
        let mut free = HashSet::default();
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    fn make(egraph: &EGraph, enode: &Lambda) -> Data {
        let f = |i: &Id| egraph[*i].data.free.iter().cloned();
        let mut free = HashSet::default();
        match enode {
            Lambda::Var(v) => { // var
                free.insert(*v);
            }
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    fn make(egraph: &EGraph, enode: &Lambda) -> Data {
        let f = |i: &Id| egraph[*i].data.free.iter().cloned();
        let mut free = HashSet::default();
        match enode {
            Lambda::Var(v) => { // var
                free.insert(*v);
            }
            Lambda::Let([v, a, b]) => { // let v = a in b(v)
                free.extend(f(b));
                free.remove(v);
                free.extend(f(a));
            }
```

Lambda calculus in egg : Analysis

```rust
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    fn make(egraph: &EGraph, enode: &Lambda) -> Data {
        let f = |i: &Id| egraph[*i].data.free.iter().cloned();
        let mut free = HashSet::default();
        match enode {
            Lambda::Var(v) => { // var
                free.insert(*v);
            }
            Lambda::Let([v, a, b]) => { // let v = a in b(v)
                free.extend(f(b));
                free.remove(v);
                free.extend(f(a));
            }
            // \v. a(v)
            Lambda::Lambda([v, a]) | Lambda::Fix([v, a]) => {
                free.extend(f(a));
                free.remove(v);
            }
            _ => enode.for_each(|c| free.extend(&egraph[c].data.free)),
        }
```

Lambda calculus in egg : Analysis

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    fn make(egraph: &EGraph, enode: &Lambda) -> Data {
        let f = |i: &Id| egraph[*i].data.free.iter().cloned();
        let mut free = HashSet::default();
        match enode {
            Lambda::Var(v) => { // var
                free.insert(*v);
            }
            Lambda::Let([v, a, b]) => { // let v = a in b(v)
                free.extend(f(b));
                free.remove(v);
                free.extend(f(a));
            }
            // \v. a(v)
            Lambda::Lambda([v, a]) | Lambda::Fix([v, a]) => {
                free.extend(f(a));
                free.remove(v);
            }
            _ => enode.for_each(|c| free.extend(&egraph[c].data.free)),
        }
        let constant = eval(egraph, enode);
        Data { constant, free }
    }
```

Lambda calculus in egg : Dynamic rewrites

```
-- 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

```
-- 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`?

Lambda calculus in egg : Dynamic rewrites

```
-- 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

```
-- 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

```
-- 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
```

How to push `let` into `lambda`?

Lambda calculus in egg : Dynamic rewrites

```
-- 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
```

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))
```

Lambda calculus in egg : Dynamic rewrites

```
-- 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
```

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))
```

Lambda calculus in egg : Dynamic rewrites

```
x_free = let v1 = e   in (\v2.  body)
x_free' = \fresh. let v1 = e   in (let v2 = fresh in body  )
```

```
rw!("let-lam-diff";
"(let ?v1 ?e (lam ?v2 ?body))" => // let v1 = e in \v2 . body
{ CaptureAvoid { // use an Applier called CaptureAvoid
    fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
    // \v2. let v1 = e in body
    if_not_free: "(lam ?v2 (let ?v1 ?e ?body))".parse().unwrap(),
    // \fresh. let v1 = e in let v2 = fresh in body
    if_free: "(lam ?fresh (let ?v1 ?e (let ?v2 (var ?fresh) ?body)))"
      .parse().unwrap(),
}}
if is_not_same_var(var("?v1"), var("?v2"))),
```

Lambda calculus in egg : Dynamic rewrites

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    ...
```

Lambda calculus in egg : Dynamic rewrites

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    ...

{ CaptureAvoid {  // rewrite pattern
    fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
    if_not_free: "(lam ?v2 (let ?v1 ?e ?body))".parse().unwrap(),
    if_free: "(lam ?fresh (let ?v1 ?e (let ?v2 (var ?fresh) ?body)))"
        .parse().unwrap(),
}}
```

Lambda calculus in egg : Dynamic rewrites

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    ...

{ CaptureAvoid {  // rewrite pattern
    fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
    if_not_free: "(lam ?v2 (let ?v1 ?e ?body))".parse().unwrap(),
    if_free: "(lam ?fresh (let ?v1 ?e (let ?v2 (var ?fresh) ?body)))"
        .parse().unwrap(),
}}

struct CaptureAvoid {
    fresh: Var, v2: Var, e: Var,
    if_not_free: Pattern<Lambda>,
    if_free: Pattern<Lambda>,
}
```

Lambda calculus in egg : Dynamic rewrites

```
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    ...
```

```
{ CaptureAvoid {  // rewrite pattern
    fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
    if_not_free: "(lam ?v2 (let ?v1 ?e ?body))".parse().unwrap(),
    if_free: "(lam ?fresh (let ?v1 ?e (let ?v2 (var ?fresh) ?body)))"
        .parse().unwrap(),
}}
```

```
struct CaptureAvoid {
    fresh: Var, v2: Var, e: Var,
    if_not_free: Pattern<Lambda>,
    if_free: Pattern<Lambda>,
}
```

```
impl Applier<Lambda, LambdaAnalysis> for CaptureAvoid {
    fn apply_one(&self, egraph: &mut EGraph, eclass: Id, subst: &Subst) ...

        let e = subst[self.e];
        let v2 = subst[self.v2];
        let v2_free_in_e = egraph[e].data.free.contains(&v2);
```

Lambda calculus in egg : Dynamic rewrites

```rust
struct Data { free: HashSet<Id>, constant: Option<Lambda>, }
impl Analysis<Lambda> for LambdaAnalysis {
    type Data = Data;
    ...
```

```rust
{ CaptureAvoid {  // rewrite pattern
    fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
    if_not_free: "(lam ?v2 (let ?v1 ?e ?body))".parse().unwrap(),
    if_free: "(lam ?fresh (let ?v1 ?e (let ?v2 (var ?fresh) ?body)))"
        .parse().unwrap(),
}}
```

```rust
struct CaptureAvoid {
    fresh: Var, v2: Var, e: Var,
    if_not_free: Pattern<Lambda>,
    if_free: Pattern<Lambda>,
}
```

```rust
impl Applier<Lambda, LambdaAnalysis> for CaptureAvoid {
    fn apply_one(&self, egraph: &mut EGraph, eclass: Id, subst: &Subst) ...

        let e = subst[self.e];
        let v2 = subst[self.v2];
        let v2_free_in_e = egraph[e].data.free.contains(&v2);

        if v2_free_in_e { ...  } else { ...  }
```

Conclusion

- Use equalities to find equivalent programs.
- Extract out the best program using a local analysis.
- Analysis + Rewrites in a single unified framework.