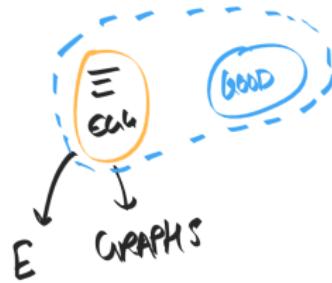


egg : Fast and extensible equality saturation

Siddharth Bhat

Monday, Jan 18 2021



How do compilers work?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}
```

How do compilers work?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    if (1 == 1) // 1. constant propagation  
        printf("foo")  
    else  
        printf("bar")  
}
```

How do compilers work?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    if (1 == 1) // 1. constant propagation  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    if (true) // 2. Canonicalization  
        printf("foo")  
    else  
        printf("bar")  
}
```

How do compilers work?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    if (1 == 1) // 1. constant propagation  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    if (true) // 2. Canonicalization  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int main() {  
    // 3. control flow simplification  
    printf("foo")  
}
```

How do compilers work?

```
int foo() {
    int x = 1;
    if (x == 1)
        printf("foo")
    else
        printf("bar")
}

int foo() {
    if (1 == 1) // 1. constant propagation
        printf("foo")
    else
        printf("bar")
}

int foo() {
    if (true) // 2. Canonicalization
        printf("foo")
    else
        printf("bar")
}

int main() {
    // 3. control flow simplification
    printf("foo")
}
```

In what order?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}
```

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what order?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    // 1. constant propagation  
    if (1 == 1)  
        printf("foo")  
    else  
        printf("bar")  
}
```

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    // 1. Constant propagation  
    int y = x == x;  
    if (y) // NOT a constant!  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what order?

```
int foo() {  
    int x = 1;  
    if (x == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    // 1. constant propagation  
    if (1 == 1)  
        printf("foo")  
    else  
        printf("bar")  
}  
  
int foo() {  
    // 2. Canonicalization  
    if (true)  
        printf("foo")  
    else  
        printf("bar")  
}
```

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    // 1. Constant propagation  
    int y = x == x;  
    if (y) // NOT a constant!  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    // 2. Canonicalization  
    int y = true;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what order?

```
int foo() {
    int x = 1;
    if (x == 1)
        printf("foo")
    else
        printf("bar")
}

int foo() {
    // 1. constant propagation
    if (1 == 1)
        printf("foo")
    else
        printf("bar")
}

int foo() {
    // 2. Canonicalization
    if (true)
        printf("foo")
    else
        printf("bar")
}

int main() {
    // 3. control flow simplification
    printf("foo")
}
```

```
int bar() {
    int y = 42 == 42;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 1. Constant propagation
    int y = x == x;
    if (y) // NOT a constant!
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 2. Canonicalization
    int y = true;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    int y = true;
    // 3. Control flow simplification
    if (y) // Cannot simplify control flow.
        printf("foo");
    else
        printf("bar");
}
```

In what Order? All of them

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what Order? All of them

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

```
int bar(int x) {  
    // 1. Constant propagation  
    int y = x == x;  
    if (y) // NOT a constant!  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what Order? All of them

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    // 2. Canonicalization  
    int y = true;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

```
int bar(int x) {  
    // 1. Constant propagation  
    int y = x == x;  
    if (y) // NOT a constant!  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what Order? All of them

```
int bar() {  
    int y = 42 == 42;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    // 2. Canonicalization  
    int y = true;  
    if (y)  
        printf("foo");  
    else  
        printf("bar");  
}
```

```
int bar(int x) {  
    // 1. Constant propagation  
    int y = x == x;  
    if (y) // NOT a constant!  
        printf("foo");  
    else  
        printf("bar");  
}  
  
int bar(int x) {  
    int y = true;  
    // 3. Control flow simplification  
    if (y) // Cannot simplify control flow.  
        printf("foo");  
    else  
        printf("bar");  
}
```

In what Order? All of them

```
int bar() {
    int y = 42 == 42;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 2. Canonicalization
    int y = true;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 4. Constant propagation
    int y = true;
    if (true) // IS a constant now!
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 1. Constant propagation
    int y = x == x;
    if (y) // NOT a constant!
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    int y = true;
    // 3. Control flow simplification
    if (y) // Cannot simplify control flow.
        printf("foo");
    else
        printf("bar");
}
```

In what Order? All of them

```
int bar() {
    int y = 42 == 42;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 2. Canonicalization
    int y = true;
    if (y)
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    // 4. Constant propagation
    int y = true;
    if (true) // IS a constant now!
        printf("foo");
    else
        printf("bar");
}
```

```
int bar(int x) {
    // 1. Constant propagation
    int y = x == x;
    if (y) // NOT a constant!
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    int y = true;
    // 3. Control flow simplification
    if (y) // Cannot simplify control flow.
        printf("foo");
    else
        printf("bar");
}

int bar(int x) {
    int y = true;
    // 5. Control flow simplification
    printf("foo");
}
```

Solutions to the phase ordering problem

Solutions to the phase ordering problem

```
bollu@cantordust:~/ > clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c
```

Solutions to the phase ordering problem

```
bollu@cantordust:~/> clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
bollu@cantordust:~/> clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
Pass Arguments: -tti -targetlibinfo -tbaa -scoped-noalias-aa  
-instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -openmpopt  
-function-attrs -argpromotion -domtree -sroa -basic-aa -aa -memoryssa  
-early-cse-memssa -aa -lazy-value-info -jump-threading -correlated-propagation  
-simplifycfg -domtree -aggressive-instcombine -basic-aa -aa -loops  
-lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine  
-libcalls-shrinkwrap -loops -postdomtree -branch-prob -block-freq  
-lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basic-aa  
-aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim  
-simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-verification  
-lcssa -basic-aa -aa -scalar-evolution -loop-rotate -memoryssa  
-lazy-branch-prob -lazy-block-freq -licm -loop-unswitch -simplifycfg -domtree  
-basic-aa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-instcombine -loop-simplify -lcssa -scalar-evolution  
-loop-idiom -indvars -loop-deletion -loop-unroll -sroa -aa -mldst-motion  
-phi-values -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-gvn -phi-values -basic-aa -aa -memdep -memcpopt -sccp -demanded-bits -bdce  
-aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine  
-lazy-value-info -jump-threading -correlated-propagation -postdomtree -adce  
-basic-aa -aa -memoryssa -dse -loops -loop-simplify -lcssa-verification -lcssa  
-aa -scalar-evolution -lazy-branch-prob -lazy-block-freq -licm -simplifycfg  
-domtree -basic-aa -aa -loops -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg  
-rpo-function-attrs -globalopt -globaldce -basiccg -globals-aa -domtree  
-float2int -lower-constant-intrinsics -domtree -loops -loop-simplify  
-lcssa-verification -lcssa -basic-aa -aa -scalar-evolution -loop-rotate  
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-loop-distribute -postdomtree -branch-prob -block-freq -scalar-evolution  
-basic-aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -inject-tli-mappings -loop-vectorize -loop-simplify  
-scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-freq  
-loop-load-elim -basic-aa -aa -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -instcombine -simplifycfg -domtree -loops -scalar-evolution  
-basic-aa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -inject-tli-mappings -slp-vectorizer -vector-combine  
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa
```

Solutions to the phase ordering problem

```
bollu@cantordust:~/ > clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c
```

```
bollu@cantordust:~/ > clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c
```

```
Pass Arguments:
```

```
-instcombine
```

```
...
```

Solutions to the phase ordering problem

```
bollu@cantordust:~/> clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
bollu@cantordust:~/> clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
Pass Arguments: -tti -targetlibinfo -tbaa -scoped-noalias-aa  
-instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -openmpopt  
-function-attrs -argpromotion -domtree -sroa -basic-aa -aa -memoryssa  
-early-cse-memssa -aa -lazy-value-info -jump-threading -correlated-propagation  
-simplifycfg -domtree -aggressive-instcombine -basic-aa -aa -loops  
-lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine  
-libcalls-shrinkwrap -loops -postdomtree -branch-prob -block-freq  
-lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basic-aa  
-aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim  
-simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-verification  
-lcssa -basic-aa -aa -scalar-evolution -loop-rotate -memoryssa  
-lazy-branch-prob -lazy-block-freq -licm -loop-unswitch -simplifycfg -domtree  
-basic-aa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-instcombine -loop-simplify -lcssa -scalar-evolution  
-loop-idiom -indvars -loop-deletion -loop-unroll -sroa -aa -mldst-motion  
-phi-values -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-gvn -phi-values -basic-aa -aa -memdep -memcpopt -sccp -demanded-bits -bdce  
-aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine  
-lazy-value-info -jump-threading -correlated-propagation -postdomtree -adce  
-basic-aa -aa -memoryssa -dse -loops -loop-simplify -lcssa-verification -lcssa  
-aa -scalar-evolution -lazy-branch-prob -lazy-block-freq -licm -simplifycfg  
-domtree -basic-aa -aa -loops -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg  
-rpo-function-attrs -globalopt -globaldce -basiccg -globals-aa -domtree  
-float2int -lower-constant-intrinsics -domtree -loops -loop-simplify  
-lcssa-verification -lcssa -basic-aa -aa -scalar-evolution -loop-rotate  
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter  
-loop-distribute -postdomtree -branch-prob -block-freq -scalar-evolution  
-basic-aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -inject-tli-mappings -loop-vectorize -loop-simplify  
-scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-freq  
-loop-load-elim -basic-aa -aa -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -instcombine -simplifycfg -domtree -loops -scalar-evolution  
-basic-aa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq  
-opt-remark-emitter -inject-tli-mappings -slp-vectorizer -vector-combine  
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa  
...
```

Solutions to the phase ordering problem

```
bollu@cantordust:~/ > clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
bollu@cantordust:~/ > clang -O3 -mllvm -debug-pass=Arguments ~/temp/foo.c  
Pass Arguments:  
-instcombine -simplifycfg
```

```
-simplifycfg  
    -instcombine
```

```
-simplifycfg  
    -instcombine
```

```
-instcombine  
    -simplifycfg  
        -instcombine
```

```
-instcombine -simplifycfg
```

```
-instcombine
```

```
...
```

Solutions? to the phase ordering problem

- ▶ $(a^*2)/2$

Solutions? to the phase ordering problem

- ▶ $(a^*2)/2$
- ▶ $(a^*2)/2$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$
- ▶ $(a*2)/2$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$
- ▶ $(a*2)/2 \xrightarrow{B} a*(2/2)$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$
- ▶ $(a*2)/2 \xrightarrow{B} a*(2/2) \rightarrow a*1$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$
- ▶ $(a*2)/2 \xrightarrow{B} a*(2/2) \rightarrow a*1 \rightarrow a$

Solutions? to the phase ordering problem

- ▶ $(a*2)/2$
- ▶ $(a*2)/2 \xrightarrow{A} (a \ll 1)/2$
- ▶ $(a*2)/2 \xrightarrow{B} a*(2/2) \rightarrow a*1 \rightarrow a$
- ▶ Rerunning (pass A; pass B) multiple times won't help. transformation loses information!

Solutions to the phase ordering problem: Equality saturation

- ▶ Is (pass A; pass B) is not the same as pass B; pass A.
- ▶ Solution: run all passes "in parallel", at the same time.

Solutions to the phase ordering problem: Equality saturation

- ▶ Is (pass A; pass B) is not the same as pass B; pass A.
- ▶ Solution: run all passes "in parallel", at the same time.
- ▶ *saturate* the original program with new programs *equivalent* to the original program.

Solutions to the phase ordering problem: Equality saturation

- ▶ Is (pass A; pass B) is not the same as pass B; pass A.
- ▶ Solution: run all passes "in parallel", at the same time.
- ▶ *saturate* the original program with new programs *equivalent* to the original program.



Figure: classical

Solutions to the phase ordering problem: Equality saturation

- ▶ Is (pass A; pass B) is not the same as pass B; pass A.
- ▶ Solution: run all passes "in parallel", at the same time.
- ▶ *saturate* the original program with new programs *equivalent* to the original program.



Figure: classical

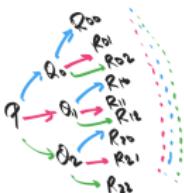


Figure: all possible transforms

Solutions to the phase ordering problem: Equality saturation

- ▶ Is (pass A; pass B) is not the same as pass B; pass A.
- ▶ Solution: run all passes "in parallel", at the same time.
- ▶ *saturate* the original program with new programs *equivalent* to the original program.



Figure: classical

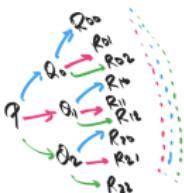
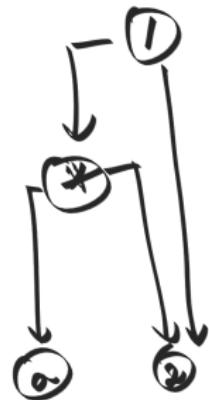


Figure: all possible transforms



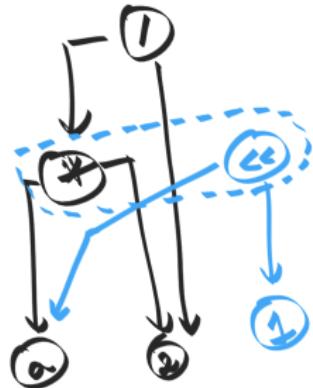
Figure: equality saturation

egg : Fast and extensible equality saturation



$(\alpha * 2) / 2 :$
 $(1 \cap \alpha, 2) \cdot 2$

egg : Fast and extensible equality saturation (2)

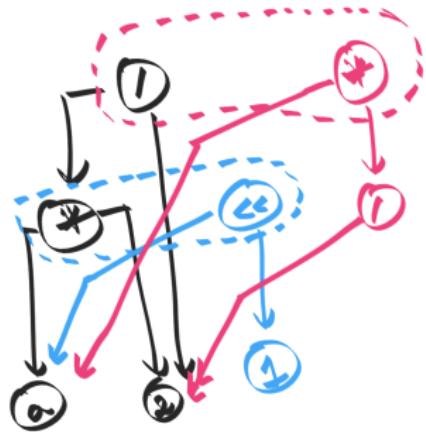


$(\alpha * 2) / 2 :$
 $(1 \cap \alpha_2), 2)$

$p * 2 \rightarrow p < 1$

$U (\alpha_2), 2)$
 $(\leq \alpha_1)$

egg : Fast and extensible equality saturation (3)



$(\alpha * 2) / 2 :$
 $(1 (* \alpha 2) 2)$

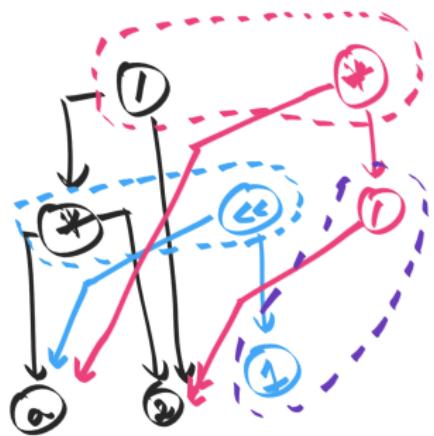
$p * 2 \rightarrow p \ll 1$

$(1 (* \alpha 2) 2)$
 $(\ll \alpha 1)$

$(p * \alpha) / 2 \rightarrow p * (\alpha / 2)$

$(1 (* \alpha 2) 2)$
 $(\frac{2}{\alpha} \alpha, 1 / 2 2)$

egg : Fast and extensible equality saturation (4)



$$\begin{array}{l} \cancel{\alpha/\alpha} \rightarrow 1 \\ (\cancel{\alpha} \cdot 1^{(2^2)}) \\ \downarrow \\ 1 \end{array}$$

$$\begin{array}{l} (\alpha*\alpha)/2: \\ (1 \cdot (\cancel{\alpha} \cdot 2)) \cdot 2 \end{array}$$

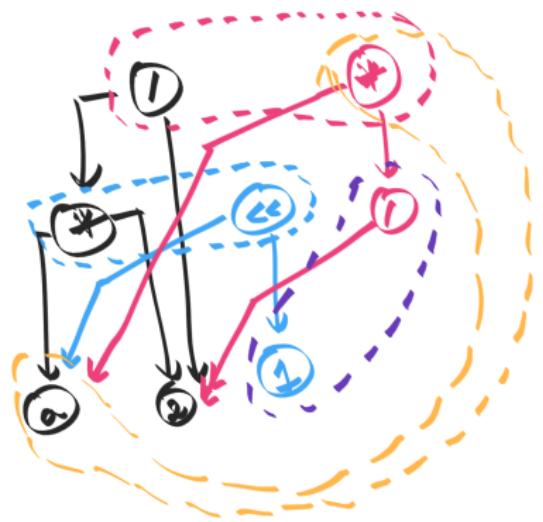
$$p*\cancel{2} \rightarrow p \ll 1$$

$$\begin{array}{l} 0 \cdot (\cancel{\alpha} \cdot 2) \cdot 2 \\ (\cancel{\alpha} \cdot \alpha) \cdot 1 \end{array}$$

$$(p*\alpha)/2 \rightarrow p*(\alpha/2)$$

$$\begin{array}{l} 0 \cdot (\cancel{\alpha} \cdot 2) \cdot 2 \\ (\cancel{\alpha} \cdot \alpha) \cdot (1 \cdot 2 \cdot 2) \end{array}$$

egg : Fast and extensible equality saturation (5)



$$x/x \rightarrow 1$$

$$(*a\{1^{22}\})$$

$$x*x \rightarrow x$$

$$(*a\{1^{22}\})$$

α

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

$$(1(*a\{2\})2)$$

$$p*p \rightarrow p \ll 1$$

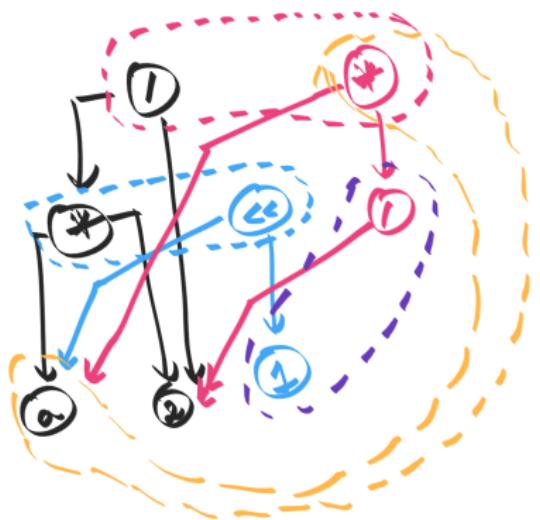
$$(1(*a\{2\})2)$$

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

$$(1(*a\{2\})2)$$

$$(*a\{1^{22}\})$$

egg : Fast and extensible equality saturation (6)



$$\alpha/\alpha \rightarrow 1$$
$$(*\alpha/(1^{22}))$$
$$\underbrace{\hspace{2cm}}_{\alpha}$$

$$(\alpha+\alpha)/2 :$$
$$(1(*\alpha/2)^2)$$

$$\alpha*\alpha \rightarrow \alpha$$
$$(*\alpha/(1^{22}))$$
$$\underbrace{\hspace{2cm}}_{\alpha}$$

$$p*2 \rightarrow p < 1$$

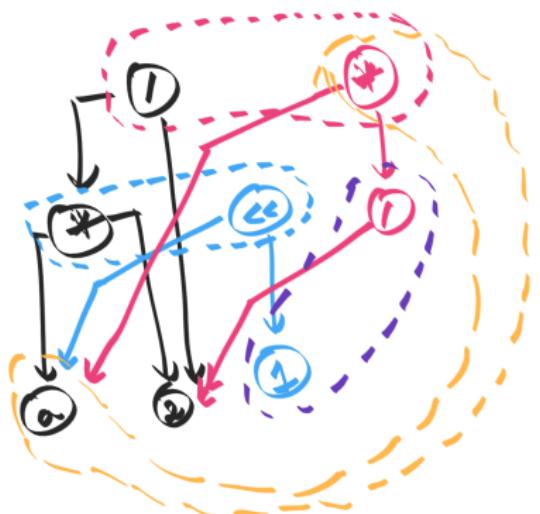
$$(1(*\alpha/2)^2)$$
$$\underbrace{\hspace{2cm}}_{(2<\alpha/2)}$$

$$(p+\alpha)/2 \rightarrow p + (\alpha/2)$$

$$(1(*\alpha/2)^2)$$
$$(*\alpha/(1^{22}))$$



egg : Fast and extensible equality saturation (7)



$x/x \rightarrow 1$

$(\alpha * \beta) / \gamma \rightarrow$

$\alpha * \beta \rightarrow \alpha$

$(\alpha * \beta) / \gamma \rightarrow$

$(\alpha * \beta) / \gamma \rightarrow$

$(1 (\alpha * \beta) \gamma)$

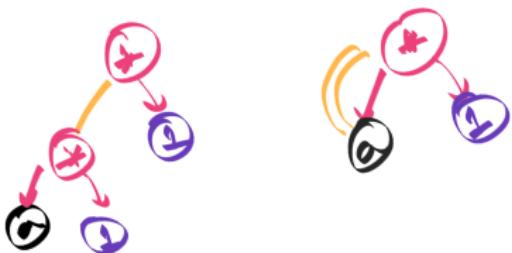
$p * 2 \rightarrow p \ll 1$

$(1 (\alpha * \beta) \gamma)$

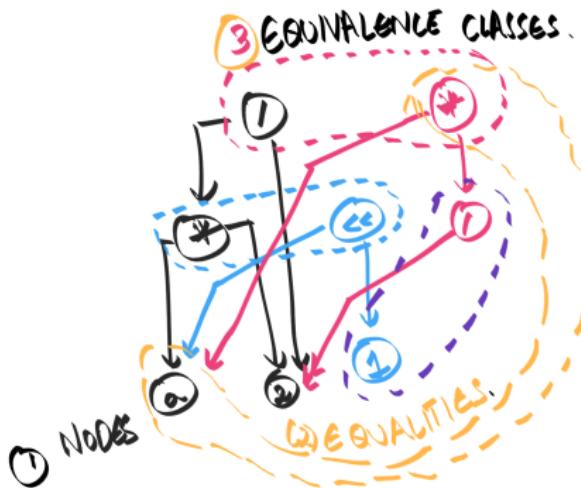
$(p * q) / \gamma \rightarrow p * (q / \gamma)$

$(1 (\alpha * \beta) \gamma)$

$(\alpha * \beta) / \gamma \rightarrow$



egg : Fast and extensible equality saturation (8)



$$x/x \rightarrow 1$$

$$(*a (*1 2)) \xrightarrow{1} 1$$

$$(\alpha * 2) / 2 : \\ (1 (* \alpha 2) 2)$$

$$x * 1 \rightarrow n$$

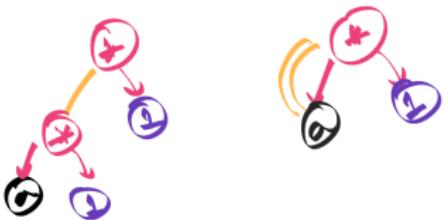
$$(*a (*1 2)) \xrightarrow{2} 2$$

$$p * 2 \rightarrow p < 1$$

$$(1 (* \alpha 2) 2) \\ (\alpha < \alpha 2)$$

$$(p * a) / 3 \rightarrow p * (a / 3)$$

$$(1 (* \alpha 2) 2) \\ (\alpha < \alpha 2)$$



Evaluation: Herbie

$$\text{sqrt}(x+1) - \text{sqrt}(x) \rightarrow 1/(\text{sqrt}(x+1) + \text{sqrt}(x))$$

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

$$\text{sqrt}(x+1) - \text{sqrt}(x) \rightarrow 1/(\text{sqrt}(x+1) + \text{sqrt}(x))$$

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 .

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

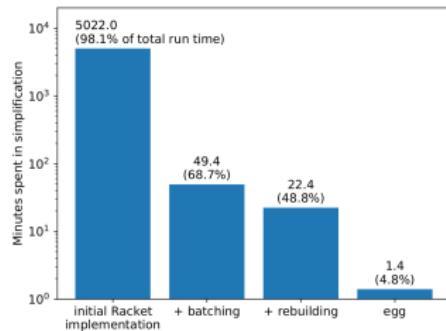
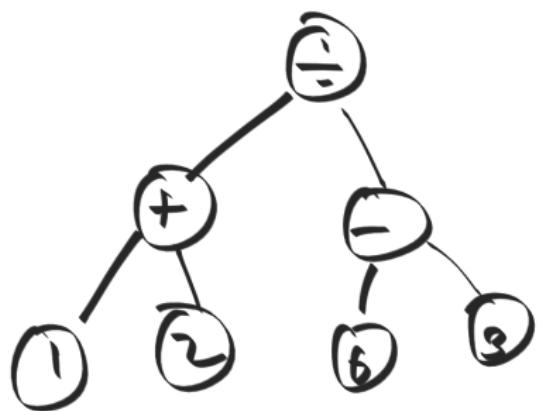


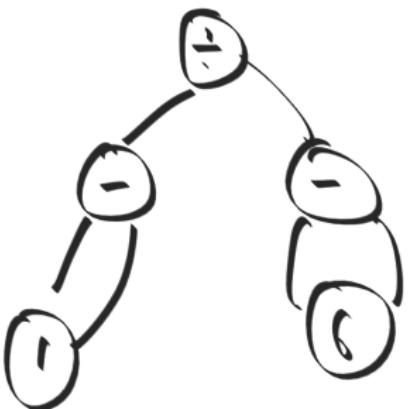
Fig. 12. Herbie sped up its expression simplification phase by adopting egg-inspired features like batched simplification and rebuilding into its Racket-based e-graph implementation. Herbie also supports using egg itself for additional speedup. Note that the y-axis is log-scale.

Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$

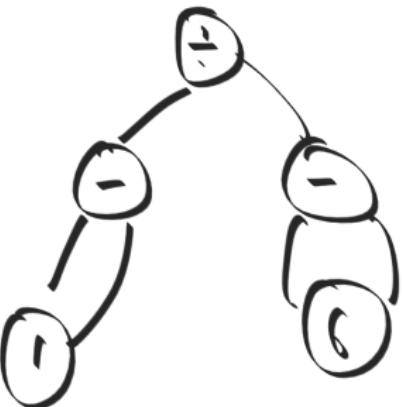
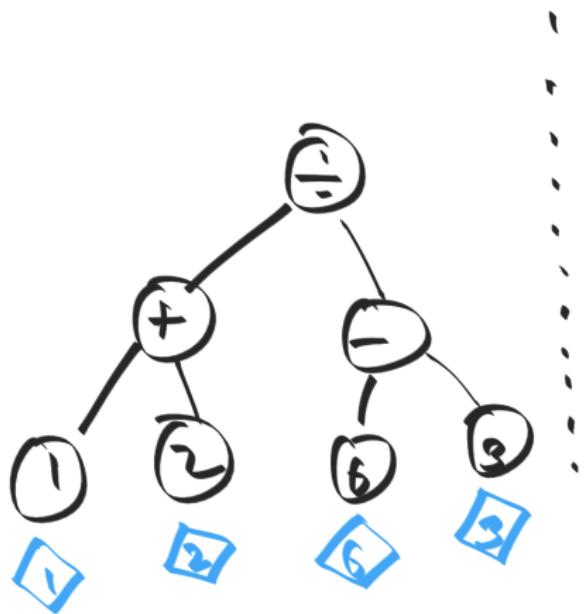


⋮



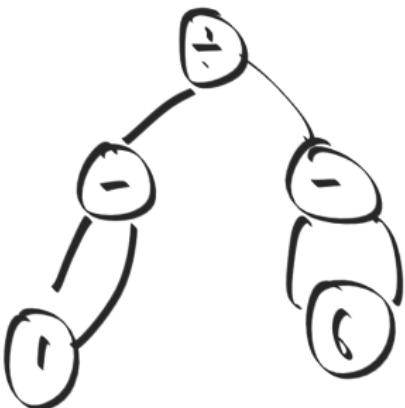
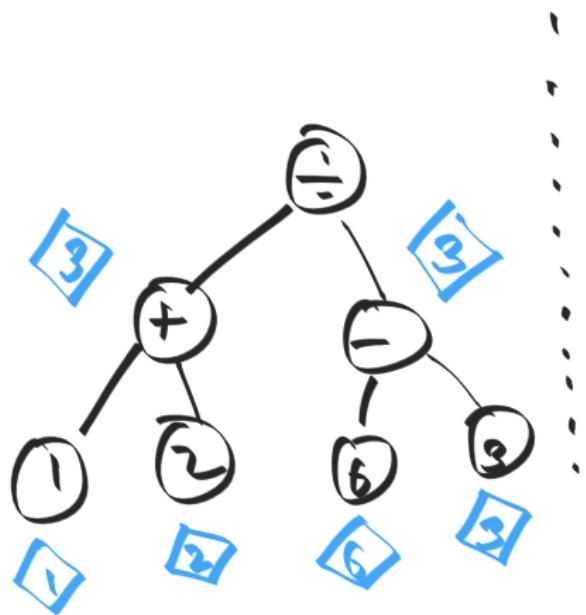
Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$



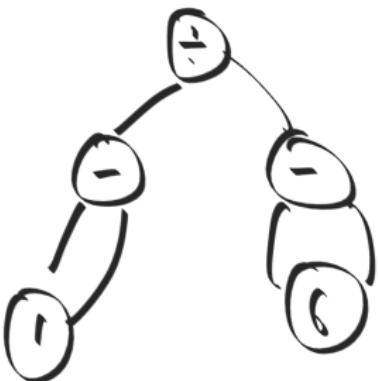
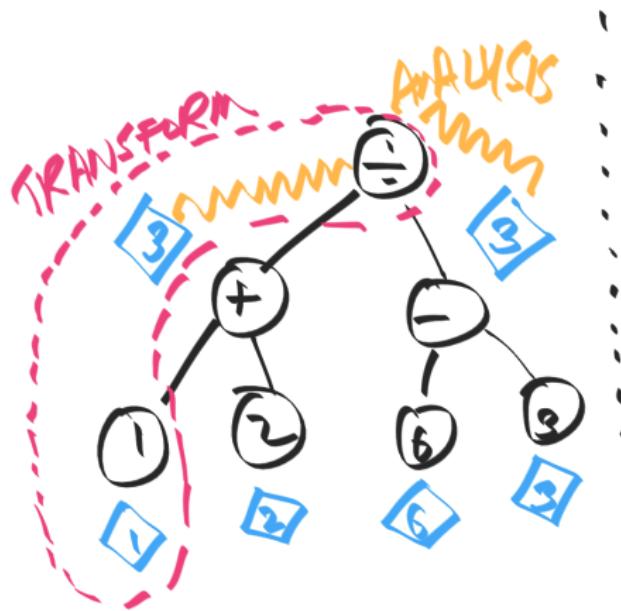
Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$



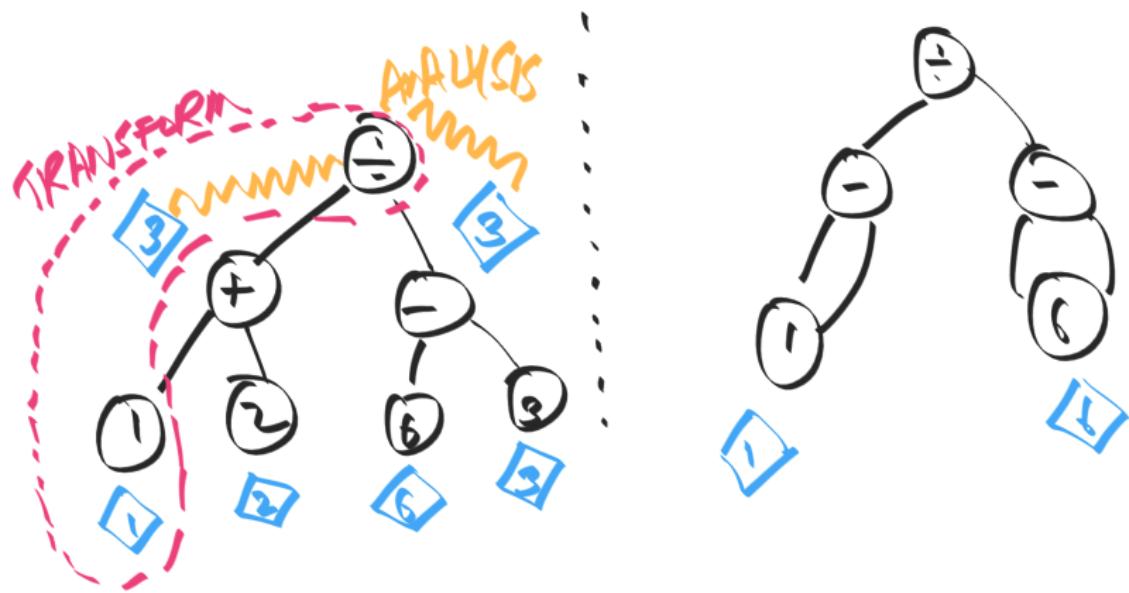
Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$



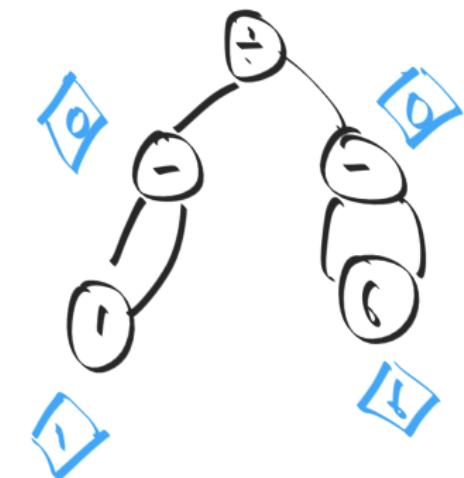
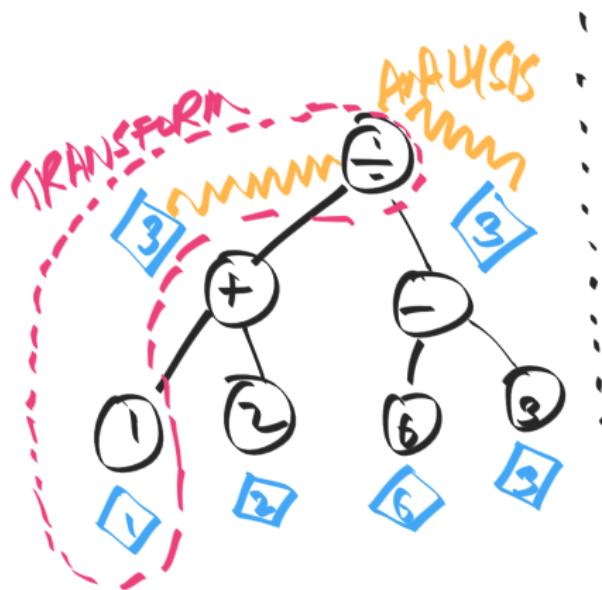
Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$



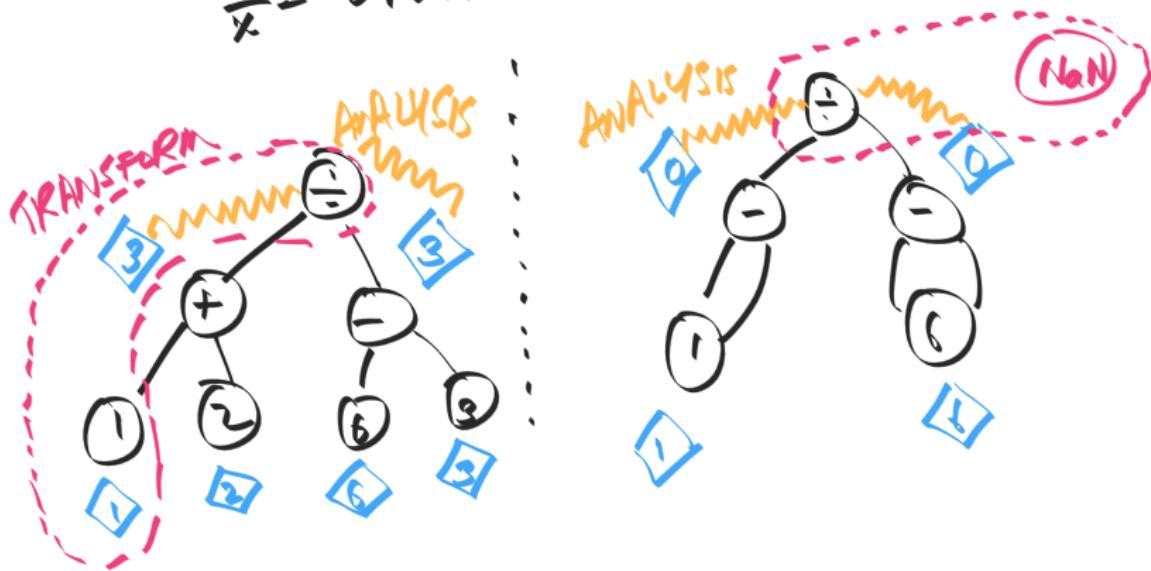
Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$



Herbie: Using egg — Analysis and rewrite

$$\frac{x}{x} = \begin{cases} 1 & : x \neq 0 \\ \text{NaN} & x = 0 \end{cases}$$

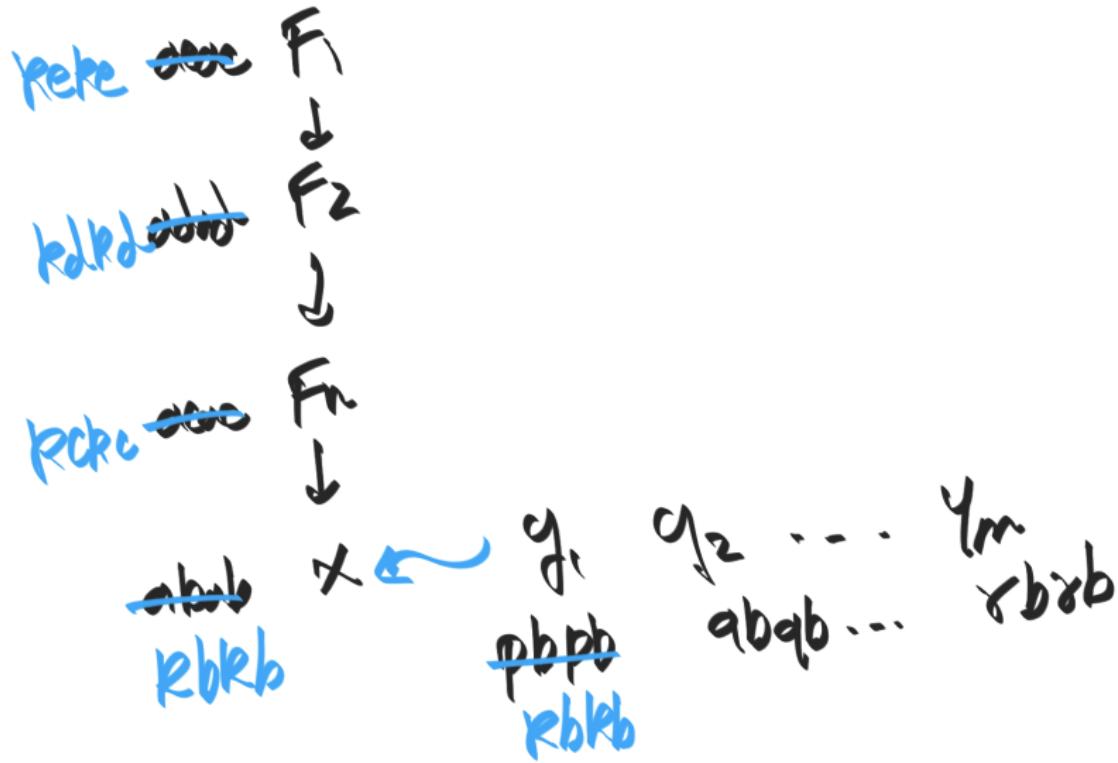


Speedup over prior art: Merging

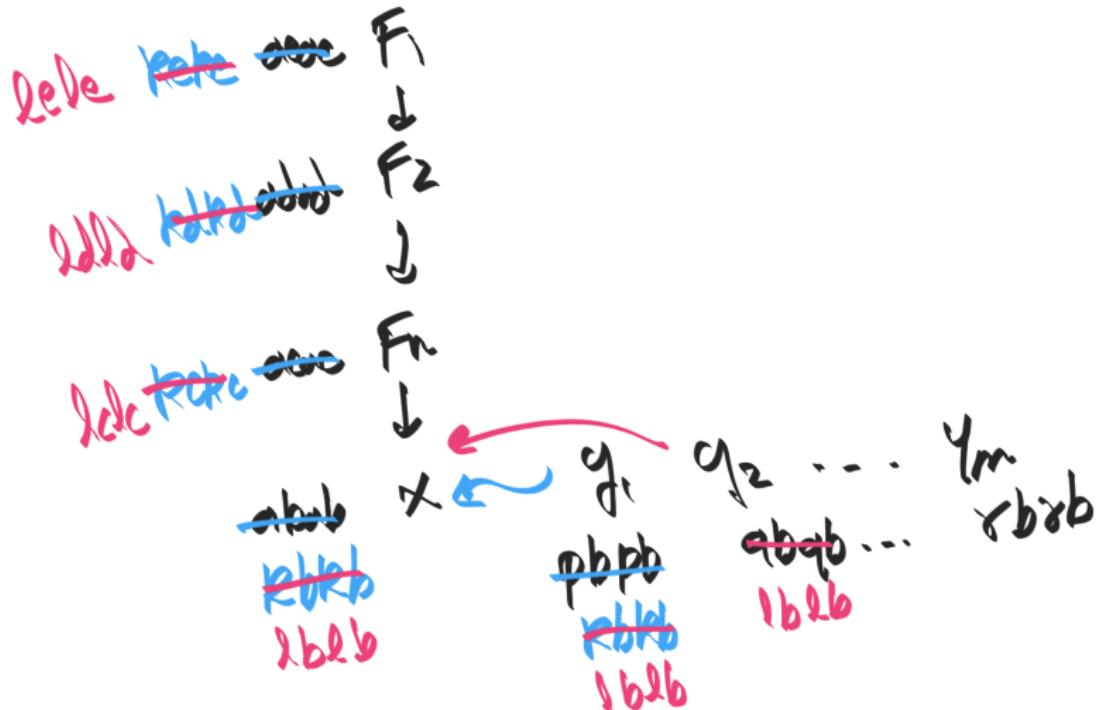
000c F_1
↓
000d F_2
↓
000c F_n
↓
abab X

$q_1 \ q_2 \ \dots \ q_m$
 $pbpb \ abqb \ \dots \ rbrb$

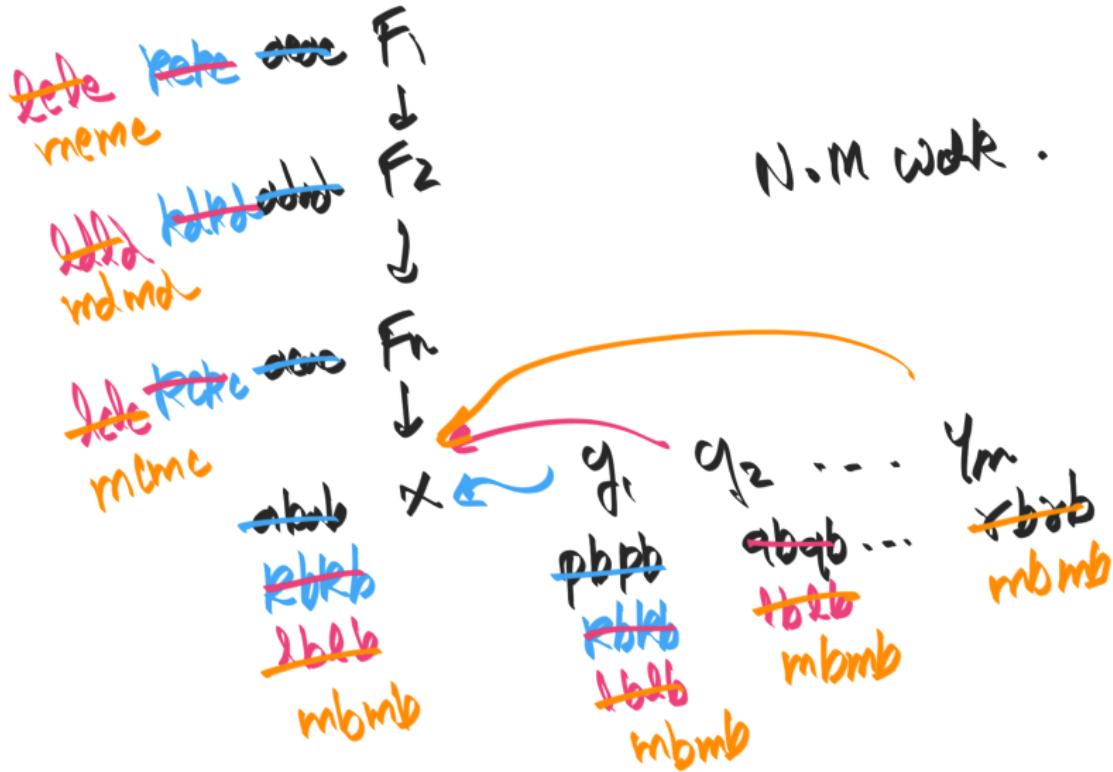
Speedup over prior art: Merging



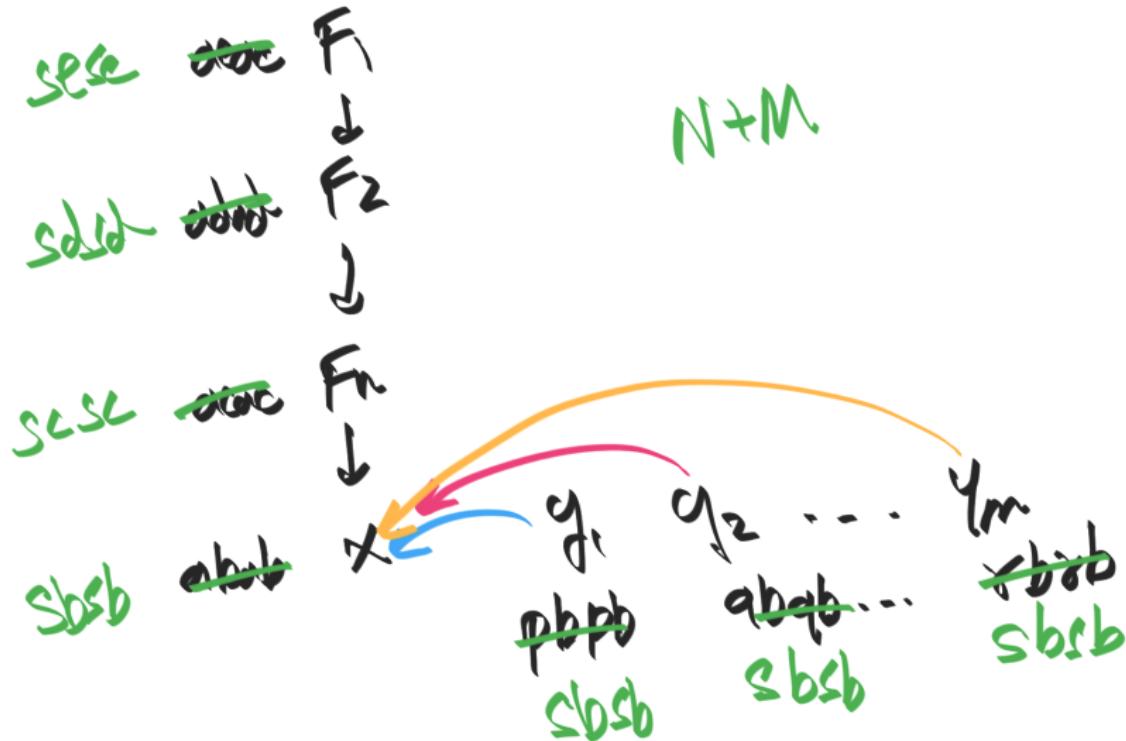
Speedup over prior art: Merging



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: $f_1(x), \dots, f_n(x), y_1, \dots, y_n$. Let the workload be $\text{merge}(x, y_1), \dots, \text{merge}(x, y_n)$. 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.

Takeaways

- ▶ Don't stick to one order; try everything at once!
- ▶ Solution to phase ordering!
- ▶ Scales reasonably well if you engineer it well
- ▶ Well-designed API that enables analysis and rewrites

Lambda calculus in egg : Dynamic rewrites redux

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

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)
```

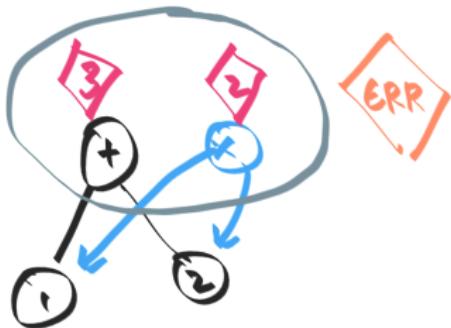
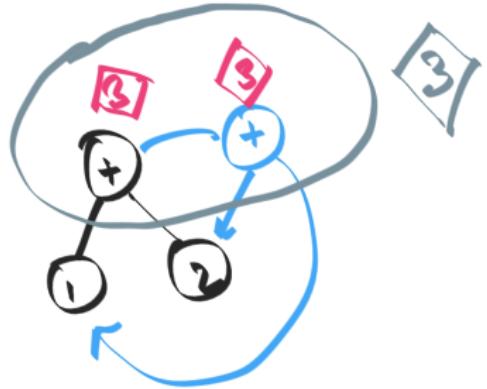
-- v2=x is free in e=x*2

```
x :: Int; x = 42
x_free'_wrong = \x. let v1 = x*2 in x + 1 ERR!
```

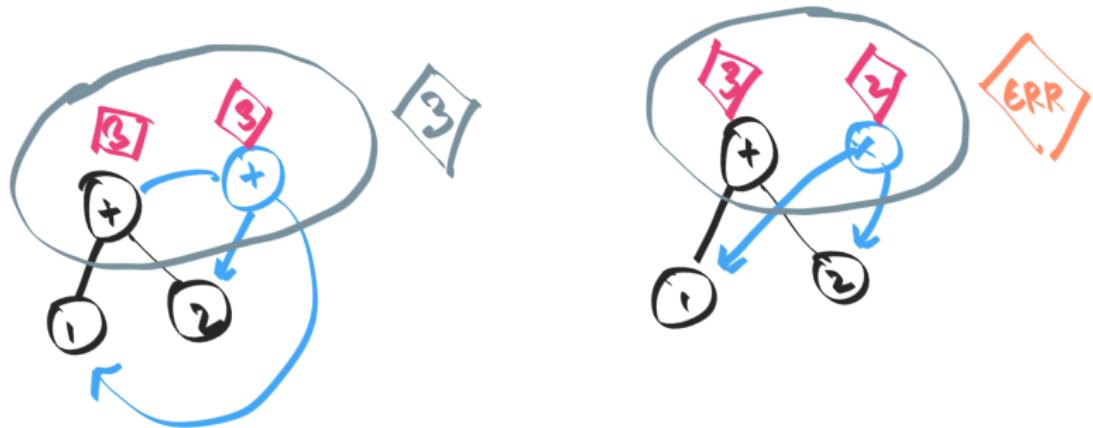
How to push let into lambda?

```
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 $\alpha : \text{node} \rightarrow L$ (Abstraction function)
- ▶ (L, \cap) is a join-semilattice.
- ▶ egg provides for each equivalence class $\text{class} \mapsto \bigcap_{\text{node} \in \text{class}} \alpha(\text{node}) \in L$