

# A Simple Introduction to Egraph

## DSA-Pre

YuMenghong

2400012921

2025 年 12 月 3 日

① Background

② Egg:Egraph-Good[WNW<sup>+</sup>21]

③ Remaining Issues

④ References

## ① Background

Origins in Congruence Closure  
Problems in Rewrite System

## ② Egg:Egraph-Good[WNW<sup>+</sup>21]

## ③ Remaining Issues

## ④ References

## ① Background

Origins in Congruence Closure

Problems in Rewrite System

## ② Egg:Egraph-Good[WNW<sup>+</sup>21]

## ③ Remaining Issues

## ④ References

# Question-1

假设我们有若干个变量符号, 表示为  $x_1, \dots, x_n$ .

现在给出若干条 (要求 Quantifier-free 的) 规则, 每条规则要么形如  $x_i = x_j$ , 要么形如  $x_i \neq x_j$ , 问是否可满足.

要求满足如下的等价关系性质:

- Reflexivity:  $x = x$
- Transitivity:  $x = y \wedge y = z \Rightarrow x = z$
- Symmetry:  $x = y \Rightarrow y = x$

# Question-1

假设我们有若干个变量符号, 表示为  $x_1, \dots, x_n$ .

现在给出若干条 (要求 Quantifier-free 的) 规则, 每条规则要么形如  $x_i = x_j$ , 要么形如  $x_i \neq x_j$ , 问是否可满足.

要求满足如下的等价关系性质:

- Reflexivity:  $x = x$
- Transitivity:  $x = y \wedge y = z \Rightarrow x = z$
- Symmetry:  $x = y \Rightarrow y = x$

答案: 使用并查集.

## Question-2

假设我们有若干个变量符号和若干个函数符号:

*FunctionSymbols* :  $f, g, h \dots$

*VariableSymbols* :  $x, y, z \dots$

*Terms* ::=  $x | f(t_1, \dots, t_n)$

现在给出若干条 (要求 Quantifier-free 的) 规则, 每条规则要么形如  $t_i = t_j$ , 要么形如  $t_i \neq t_j$ , 问是否可满足. 要求满足:

- Reflexivity:  $x = x$
- Transitivity:  $x = y \wedge y = z \Rightarrow x = z$
- Symmetry:  $x = y \Rightarrow y = x$
- Congruence:

$$t_1 = s_1 \wedge \dots \wedge t_n = s_n \Rightarrow f(t_1, \dots, t_n) = f(s_1, \dots, s_n).$$

思路: 仍然尝试求出等于关系的一个 Congruence Closure.

# Abstract Syntax Tree

AST 中, 对每个节点来说, 以它为根的一棵子树代表了一个 Term.

AST 中可能出现若干个节点的"名字"一样, 但它们表示的子树不同, 因此表示了不同的 term.

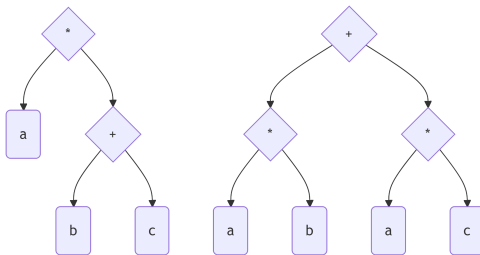


图 1: 分配律的两棵语法树

# Origin-Egraph[NO80]

尝试将并查集的思路挪到 AST 上. 如果在 AST 上表示一个 term 变成了一棵子树的根节点, 能不能对此进行图上的合并呢?

考虑以下两个操作:

- add: 向图中添加一棵 AST.(如果这棵 AST 的某个 subtree 已经在图中, 则直接连线过去, 不重复添加)
- merge: 认为以两个节点  $x, y$  表示的 term 相等, 将它们所在的并查集起来. 此时, 如果存在一个函数  $f$ , 使得  $f(x)$  和  $f(y)$  在图上都存在 (然而, 它们应当表示了不同的 term), 就尝试向上合并.

容易发现, 上述过程的确维护了在 AST 上的 term 的等价关系. 而且由于严格根据等价关系的方法构造, 它也应该是闭的 (不存在能由上述推出的等价关系不含于图中).

## Origin-Egraph[NO80]: Example1

- $f(f(a)) = a$ .
- $f(f(f(f(f(a)))))) = a$ .

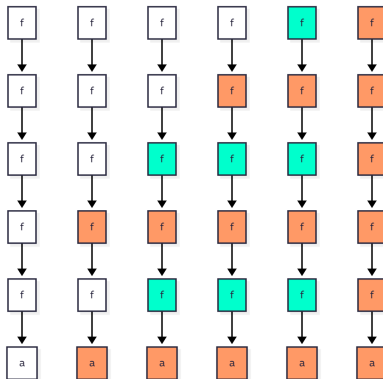


图 2: Egraph 过程 (同色代表被 union 到了一起)

# Some Problems

然而, 此时的 Egraph 还是存在一些问题:

- 当图上存在  $f(a)$  以及  $a = b$  时, 图上并没有显式地展示  $f(a) = f(b)$ , 因为  $f(b)$  并没有被加入图中 (除非你去手动加入一个  $f(b)$ , 然后再向上传播).
- 在上一条的基础上, 以上面的例子为例, Egraph 并不能显式表示无穷. 例如在上面的图中不存在一个  $f(\dots f(a))$  嵌套无穷多次的 term.
- 重写规则还只能是 Quatifier-free 的.
- 当一个  $f(t_1, \dots, t_n)$  的参数非常多的时候, 每次合并其中一个参数都需要向上传播一次, 这个过程相当耗费时间.



# Problems in Rewrite System

在编译器优化的时候, 我们可能希望它将复杂的运算改为简单且快速的操作 (Strength Reduction), 其中一个重要的工具是通过基于重写 (Rewrite) 的优化, 我们可能希望编译器做到 (不考虑精度问题):

- $x * 2 \Rightarrow x \ll 1$
- $(x * 2) / 2 \Rightarrow x$

# Problems in Rewrite System

在编译器优化的时候, 我们可能希望它将复杂的运算改为简单且快速的操作 (Strength Reduction), 其中一个重要的工具是通过基于重写 (Rewrite) 的优化, 我们可能希望编译器做到 (不考虑精度问题):

- $x * 2 \Rightarrow x << 1$
- $(x * 2) / 2 \Rightarrow x$

为了实现第一个目标, 我们可能会尝试去模式匹配 (pattern matching) 代码中所有形如  $t * 2$  的表达式, 并将其重写为  $t << 1$ . 为了实现第二个目标, 我们可能会先告诉编译器, 可以将  $(t * a) / b$  优化为  $t * (a / b)$ , 然后再有  $2 / 2 = 1$  以及  $t * 1 = t$ .

# Problems in Rewrite System

在编译器优化的时候, 我们可能希望它将复杂的运算改为简单且快速的操作 (Strength Reduction), 其中一个重要的工具是通过基于重写 (Rewrite) 的优化, 我们可能希望编译器做到 (不考虑精度问题):

- $x * 2 \Rightarrow x \ll 1$
- $(x * 2)/2 \Rightarrow x$

为了实现第一个目标, 我们可能会尝试去模式匹配 (pattern matching) 代码中所有形如  $t * 2$  的表达式, 并将其重写为  $t \ll 1$ . 为了实现第二个目标, 我们可能会先告诉编译器, 可以将  $(t * a)/b$  优化为  $t * (a/b)$ , 然后再有  $2/2 = 1$  以及  $t * 1 = t$ . 然而, 重写优化往往是破坏性 (destructive) 的. 例如  $(x * 2)/2$ , 它会首先将其重写为  $(x \ll 1)/2$ , 然后就无法匹配到第二个目标了. 这样我们留下的项仍然不是我们心目中的最优解 (单独的  $x$ ). 究竟应该以何种顺序 rewrite, 这个问题一般被叫做 phase ordering problem.

# Equality Saturation

Egraph 的优势:

- Egraph 可以维护任意 AST, 甚至可以有循环, 泛用性足够广.
- Egraph 可以保留"重写前"的信息, 在 Egraph 上做重写不是破坏性的.

借助 Egraph 和若干条预定 rewrite 规则, 我们可以找到一个 term 的若干等价形式, 并抽出其中最优的那个.

这个过程一般被叫做 Equality Saturation.

# Equality Saturation

Egraph 的优势:

- Egraph 可以维护任意 AST, 甚至可以有循环, 泛用性足够广.
- Egraph 可以保留"重写前"的信息, 在 Egraph 上做重写不是破坏性的.

借助 Egraph 和若干条预定 rewrite 规则, 我们可以找到一个 term 的若干等价形式, 并抽出其中最优的那个.

这个过程一般被叫做 Equality Saturation.

然而, 当我们写代码的时候, 写出的往往不是 Rule, 而是一个 Interpreter 负责将 Syntax 翻译到 Semantics. 如何对于一个 Interpreter, 尝试去求出若干"可能"有价值的 Rule 呢?

# Rewrite Rule Inference[NWZ<sup>+</sup>21]

传统的解决方案是, 先暴力枚举一些项, 然后将它们两两配对, 并使用外部证明工具 (如 SMT) 暴力判断它们是否相等.

这个过程可能会出现非常多浪费. 例如先浪费大量时间证明了  $a + a = a * 2$ , 又浪费大量时间证明了  $(a + a) + 1 = (a * 2) + 1$ .

# Rewrite Rule Inference [NWZ<sup>+</sup>21]

传统的解决方案是, 先暴力枚举一些项, 然后将它们两两配对, 并使用外部证明工具 (如 SMT) 暴力判断它们是否相等。

这个过程可能会出现非常多浪费. 例如先浪费大量时间证明了  $a + a = a * 2$ , 又浪费大量时间证明了  $(a + a) + 1 = (a * 2) + 1$ . 如何减少这种浪费? 仍然可以使用 Egraph.

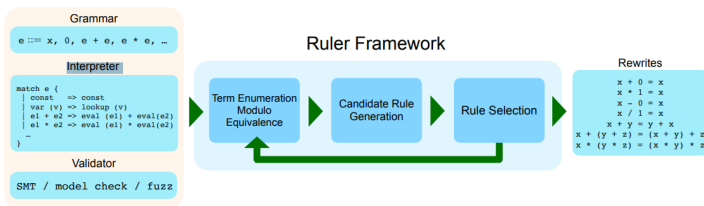


图 3: Rewrite Rule Inference

## 1 Background

## 2 Egg:Egraph-Good[WNW<sup>+</sup>21]

Refinement

Rebuilding

E-class Analysis

## 3 Remaining Issues

## 4 References



# New Syntax

egg 使用了新的 Syntax:

function symbols  $f, g$

e-class ids  $a, b$

terms  $t ::= f \mid f(t_1, \dots, t_m)$

e-nodes  $n ::= f \mid f(a_1, \dots, a_m)$

e-classes  $c ::= \{n_1, \dots, n_m\}$

opaque identifiers

$m \geq 1$

$m \geq 1$

$m \geq 1$

图 4: Syntax

# Hascons Invariant

egg 使用一个三元组  $(U, M, H)$  来维护一个 Egraph, 其中:

- $U$  是一个并查集, 处理 e-class ids 之间的等价关系<sup>1</sup>
- $M$  是一个将 e-class id 映射到 e-class 的函数
- $H$  是一个从 e-node 映射到 e-class id 的函数

对于一个 e-node, 定义

$$\text{canon}(f(a_1, \dots, a_m)) = f(\text{find}(a_1), \dots, \text{find}(a_m)).$$

它们应当满足: 对于一个 e-node  $n$  和一个 e-class id  $a$ , 恒有  
 $n \in M[a] \Leftrightarrow H[\text{canon}(n)] = \text{find}(a)$ .

---

<sup>1</sup>一个令人难过的事实是, 由于合并后还要向上传播, 启发式合并的技巧在这里没用了

# Representation of Terms

我们定义:

- 一个 e-graph 能表示一个 term 当且仅当它的某个子 e-class 能表示
- 一个 e-class 能表示一个 term 当且仅当它的某个子 e-node 能表示
- 一个 e-node  $f(a_1, \dots, a_m)$  能表示一个 term  $f(t_1, \dots, t_m)$ , 当且仅当  $\forall i, a_i$  所表示的 e-class 能表示  $t_i$ .

# Representation of Terms

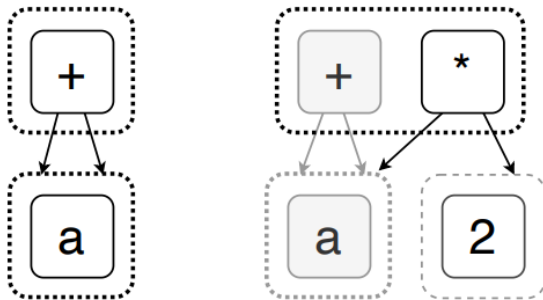
我们定义:

- 一个 e-graph 能表示一个 term 当且仅当它的某个子 e-class 能表示
- 一个 e-class 能表示一个 term 当且仅当它的某个子 e-node 能表示
- 一个 e-node  $f(a_1, \dots, a_m)$  能表示一个 term  $f(t_1, \dots, t_m)$ , 当且仅当  $\forall i, a_i$  所表示的 e-class 能表示  $t_i$ .

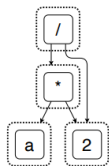
我们可以在上述基础上定义 Equivalence 符号:

- 对于两个 e-class ids: 定义  $a \equiv_{id} b$  当且仅当它们代表了同一个 e-class, 或说  $find(a) = find(b)$
- 对于两个 e-nodes: 定义  $n_1 \equiv_{node} n_2$  当且仅当  $n_1, n_2$  在同一个 e-class 里
- 对于两个 terms: 定义  $t_1 \equiv_{term} t_2$  当且仅当  $t_1, t_2$  能被同一个 e-class 表示

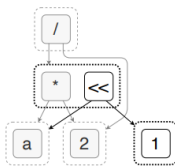
## Example1

图 5: Example( $a + a = a * 2$ )

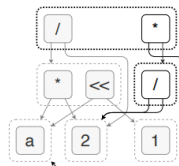
## Example2



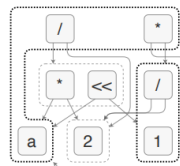
(a) Initial e-graph contains  $(a \times 2)/2$ .



(b) After applying rewrite  $x \times 2 \rightarrow x \ll 1$ .



(c) After applying rewrite  $(x \times y)/z \rightarrow x \times (y/z)$ .



(d) After applying rewrites  $x/x \rightarrow 1$  and  $1 \times x \rightarrow x$ .

图 6: Example( $((a \times 2)/2 = a)$ )

## Example3-Group

一个群的初始定义为:

- 存在单位元  $0$ , 使得  $\forall x, x + 0 = x = 0 + x$ .
- 具有结合律,  $\forall xyz, (x + y) + z = x + (y + z)$ .
- 存在逆元,  $\forall x, \text{inv}(x) + x = 0$ .

尝试以此证明:  $\text{inv}(\text{inv}(x)) = x$ .

## Example3-Group

一个群的初始定义为:

- 存在单位元  $0$ , 使得  $\forall x, x + 0 = x = 0 + x$ .
- 具有结合律,  $\forall xyz, (x + y) + z = x + (y + z)$ .
- 存在逆元,  $\forall x, \text{inv}(x) + x = 0$ .

尝试以此证明:  $\text{inv}(\text{inv}(x)) = x$ .

$$\begin{aligned}\text{inv}(\text{inv}(x)) &= \text{inv}(\text{inv}(x)) + 0 && [\text{unit}] \\ &= \text{inv}(\text{inv}(x)) + (\text{inv}(x) + x) && [\text{inv}] \\ &= (\text{inv}(\text{inv}(x)) + \text{inv}(x)) + x && [\text{assoc}] \\ &= 0 + x && [\text{inv}] \\ &= x && [\text{unit}]\end{aligned}$$

## Example3-Group

尝试证明: $\text{inv}(\text{inv}(x)) = x$ .

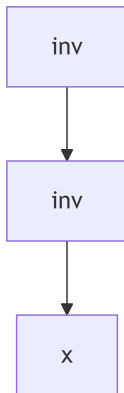


图 7: 初始

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

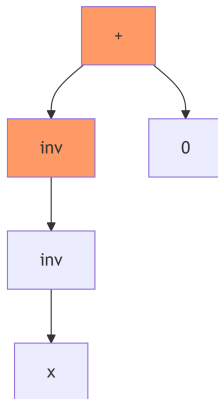


图 7: 单位元规则

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

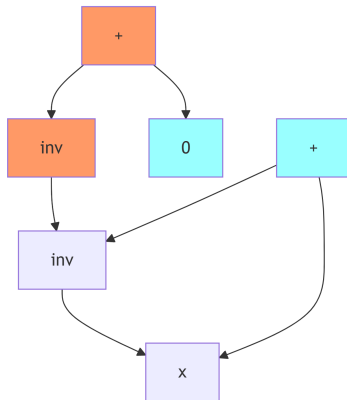


图 7: 逆元规则

## Example3-Group

尝试证明: $\text{inv}(\text{inv}(x)) = x$ .

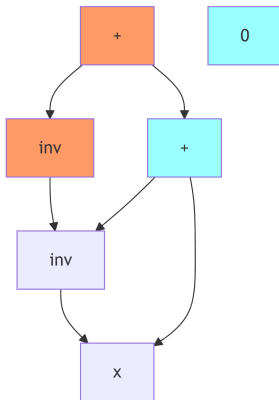


图 7: 逆元规则

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

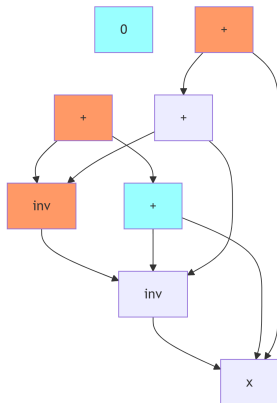


图 7: 结合律规则

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

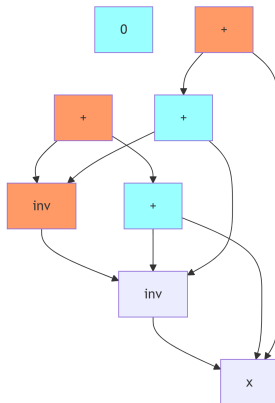


图 7: 逆元规则

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

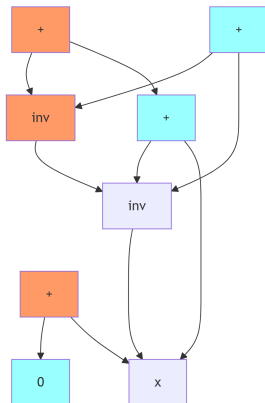


图 7: 逆元规则

## Example3-Group

尝试证明:  $\text{inv}(\text{inv}(x)) = x$ .

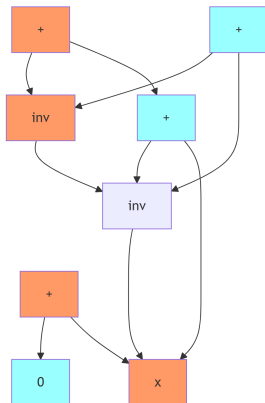


图 7: 单位元规则

## 1 Background

## 2 Egg:Egraph-Good[WNW<sup>+</sup>21]

Refinement

Rebuilding

E-class Analysis

## 3 Remaining Issues

## 4 References

# Rebuilding

简而言之: 当我们引入一条等价关系 (或说: 合并两个 e-class) 的时候, 并不立刻向上传播. 而是等最后统一向上合并.

## 1 Background

## 2 Egg:Egraph-Good[WNW<sup>+</sup>21]

Refinement

Rebuilding

E-class Analysis

## 3 Remaining Issues

## 4 References

# Motivation

Question1: 有一些自然的优化不应该需要特别的 Rewrite Rules, 比如 Constant folding.

Question2: 有一些 Rewrite 不是无条件进行的, 比如  $x/x \rightarrow 1$  必须保证  $x \neq 0$ .

Question3: 在 Program Optimization 时, 我们最后需要从 Egraph 中提取出一个最优的 term(计算时间最小), 然而, 最后统一遍历整张图的代价往往较大.

Solution: 尝试在 Egraph 的建图过程中, 就自动维护一些信息.

## Additional Information

考虑什么样的信息是 Egraph 可以维护. 首先它应当是 local 的, 也就是一个 e-node  $f(a_1, \dots, a_m)$  的信息只由  $f$  的信息和 e-class  $a_1, \dots, a_m$  各自的信息计算得到; 并且一个 e-class 的信息应该只由其包含的 e-node 的信息以某种方式"合并"起来. 此外,(我认为) $f(a_1, \dots, a_m)$  的信息应该"多于" $a_1, \dots, a_m$  的信息 (在下一页表现为偏序关系).

此外, 为了方便我们维护, 它应当可以进行如下操作:

- make: 当 Add 一个新的 e-node 的时候, 可以生成一个新的信息.
- join: 当 Union 两个 e-class 的时候, 可以合并它们的信息.
- modify: 当一个 e-class 的某个子 e-node 的信息改变的时候, 可以快速更新它的信息 (这里依赖于 Rebuilding).

# Join-Semilattice

我们断言: 满足 Join-Semilattice 关系的 local 的信息是可维护的.  
一个 Join-Semilattice 定义为一个偏序集  $(P, \geq)$ , 其中任意两个元素  $x, y \in P$ , 都存在一个运算  $x \wedge y \in P$ , 使得  $x, y \geq (x \wedge y)$ , 并且满足以下性质:

- Associative:  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- Commutative:  $x \wedge y = y \wedge x$
- Idempotent :  $x \wedge x = x$

# Constant Folding

定义信息为 `Option(Number)` 类型, 其中  $a > b$  当且仅当  $b$  是 `None`. 那么我们就可以在 Egraph 的过程中做 Constant Folding 检查:

- **make:** 当 Add 一个新的表示某个常数  $c$  的 e-node 的时候, 其信息为 `Some(c)`.
- **join:** 当 Union 两个 e-class 的时候, 如果它们都不是 `None` 而且都是相同的 `Some(c)`, 就返回这个 `Some(c)`; 否则返回 `None`.
- **modify:** 当一个 e-class 的某个子 e-node 的信息改变的时候, 只需要检查它的信息就可以更新.

# Extraction

当 cost 是 local 的时候, 它就可以定义为一个 semilattice, 直接定义信息为该 cost, 其中  $a \geq b$  当且仅当  $a$  的时间消耗大于  $b$ , 这样就可以定义  $a \wedge b = \min(a, b)$ . 现在我们就可以在 Egraph 的过程中提取时间信息:

- make: 当 Add 一个新的表示某个 e-node 的时候, 直接取其 cost.
- join: 当 Union 两个 e-class 的时候, 取它们 cost 的 min.
- modify: 当一个 e-class 的某个子 e-node 的信息改变的时候, 由于该改变是由上述两个引起的, 这只会变小, 因此也可以快速更新.

- ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺ ↻

## combinatorial explosion

对于一些简单的规则 (比如交换律, 结合律, 分配律), 它们的两侧 AST 可能相差甚远. 为了 Rewrite 它们就势必需要把对应的 AST 全部添加进来.

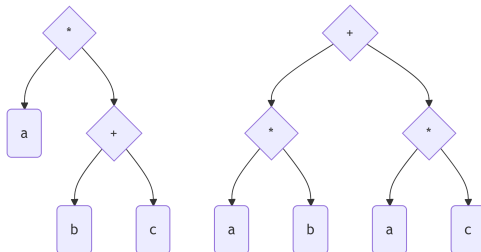


图 8: 分配律的两棵语法树

① Background

② Egg:Egraph-Good[WNW<sup>+</sup>21]

③ Remaining Issues

④ References

- [NO80] Greg Nelson and Derek C. Oppen.  
Fast decision procedures based on congruence closure.  
*J. ACM*, 27(2):356–364, April 1980.
- [NWZ<sup>+</sup>21] Chandrakana Nandi, Max Willsey, Amy Zhu,  
Yisu Remy Wang, Brett Saiki, Adam Anderson,  
Adriana Schulz, Dan Grossman, and Zachary Tatlock.  
Rewrite rule inference using equality saturation.  
*Proc. ACM Program. Lang.*, 5(OOPSLA), October  
2021.
- [WNW<sup>+</sup>21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang,  
Oliver Flatt, Zachary Tatlock, and Pavel Panchekha.  
egg: Fast and extensible equality saturation.  
*Proceedings of the ACM on Programming Languages*,  
5(POPL):1–29, January 2021.

*Thanks!*