# Optimising Functional Programs with Equality Saturation

Thomas Kœhler Michel Steuwer

University of Glasgow

The University of Edinburgh

Scottish Programming Languages Seminar — June 2021

# Term Rewriting

Which rewrite rule should be applied when, and where?

Desired optimisation:

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

Wrong turn:

$$(a * 2)/2 \longrightarrow (a \ll 1)/2$$

Infinite loop:

$$(a * 2)/2 \longrightarrow (2 * a)/2 \longrightarrow (a * 2)/2 \longrightarrow \cdots$$

# Equality Saturation

Which rewrite rule should be applied when, and where?

Explore all possibilities

[Tate et al. 2009 “Equality saturation: a new approach to optimization”]
[Willsey et al. 2021 “egg: fast and extensible equality saturation”]

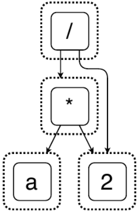
\+ No need to decide which rewrite to apply next,
Decide which program variant you want in the end.

\- Need to efficiently represent and rewrite many programs.

# Equality Saturation

## E-Graphs

$$(a * 2)/2 \longrightarrow^{*} a$$

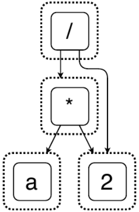


$(a * 2)/2$

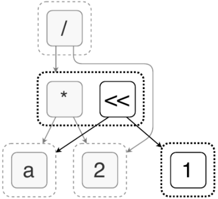
# Equality Saturation

## E-Graphs

$$(a * 2)/2 \longrightarrow^{*} a$$



$(a * 2)/2$

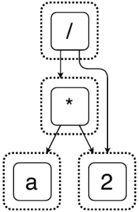


$x * 2 \longrightarrow x \ll 1$

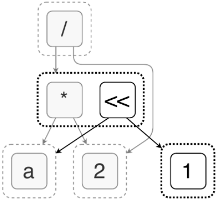
# Equality Saturation

## E-Graphs

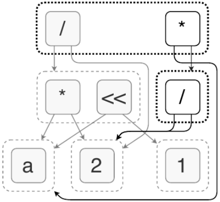
$$(a * 2)/2 \longrightarrow^{*} a$$



$(a * 2)/2$



$x * 2 \longrightarrow x \ll 1$

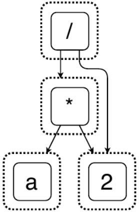


$(x * y)/z \longrightarrow x * (y/z)$

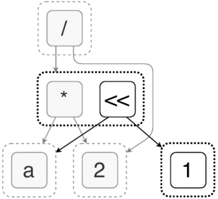
# Equality Saturation

## E-Graphs

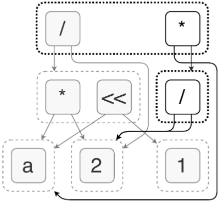
$$(a * 2)/2 \longrightarrow^{*} a$$



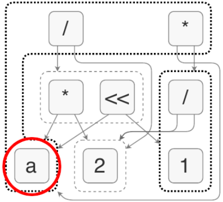
$(a * 2)/2$



$x * 2 \longrightarrow x \ll 1$



$(x * y)/z \longrightarrow x * (y/z)$



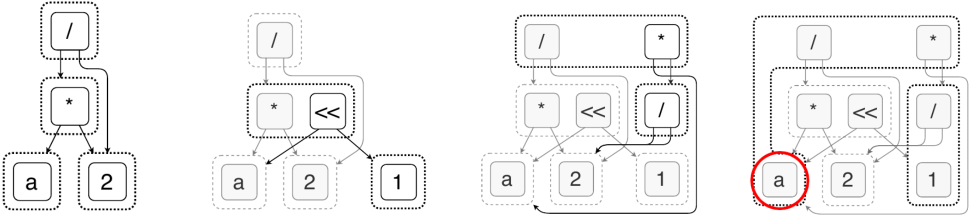
$x/x \longrightarrow 1$

$x * 1 \longrightarrow x$

# Equality Saturation

## E-Graphs

$$(a * 2)/2 \longrightarrow^{*} a$$



How does it work for functional programs?

# Equality Saturation for Functional Programs?

Rewriting RISE programs (typed lambda calculus with computational patterns):

$$(\lambda x.\ b)\,e \longrightarrow b[e/x] \qquad\qquad (\beta\text{-reduction})$$

$$\lambda x.\ f\,x \longrightarrow f \qquad \text{if } x \text{ not free in } f \qquad (\eta\text{-reduction})$$

$$map\,f\,(map\,g\ arg) \longrightarrow map\ (\lambda x.\ f\,(g\ x))\ arg \qquad\qquad (\text{map-fusion})$$

$$map\ (\lambda x.\ f\,gx) \longrightarrow \lambda y.\ map\,f\,(map\ (\lambda x.\ gx)\ \ y) \qquad \text{if } x \text{ not free in } f \qquad (\text{map-fission})$$

How can we implement substitution, predicates and name bindings?

# Substitution

$$(\lambda x.\ b)e \longrightarrow b[e/x] \qquad (\beta\text{-reduction})$$

- Small-step explicit substitution: simple but inefficient
  - overloads the e-graph with intermediate steps
- Big-step substitution over the e-graph: open challenge
- Extraction-based substitution: efficient but approximated
  1. extract terms from the e-classes $b$ and $e$
  2. perform a standard term substitution
  3. insert the result back into the e-graph

# Substitution

$$(\lambda x.\ b)\,e \longrightarrow b[e/x] \qquad (\beta\text{-reduction})$$

- Small-step explicit substitution: simple but inefficient
  - (1) is not found: out of memory after multiple seconds X
- Extraction-based substitution: efficient but approximated
  - (1) is found in milliseconds ✓

$$map\ (\lambda x.\ f_4\ (f_3\ (f_2\ (f_1\ x)))) \longrightarrow^* \lambda y.\ map\ (\lambda x.\ f_4\ (f_3\ x))\ (map\ (\lambda x.\ f_2\ (f_1\ x))\ y) \qquad (1)$$

# Predicates

$$\lambda x.\, f\, x \longrightarrow f \quad \text{if } x \text{ not free in } f \qquad (\eta\text{-reduction})$$

- if $\forall t \in f.\ x$ not free in $t$: ignores valid terms
- if $\exists t \in f.\ x$ not free in $t$: accepts invalid terms
- filter $f$ into $f_2 = \{t \mid t \in f, x \text{ not free in } t\}$: open challenge

# Name Bindings

$$map\ f\ (map\ g\ arg) \longrightarrow map\ (\lambda x.\ f\ (g\ x))\ arg \qquad \text{(map-fusion)}$$

- Fresh name on every rewrite: inefficient
  - equality modulo alpha renaming is not built-in
  - generates more and more alpha-equivalent programs
- DeBruijn indices: significant improvement
  - but no equality modulo alpha renaming for sub-terms
- To investigate: build alpha-equivalence into the e-graph
  [Maziarz et al. 2021 “Hashing modulo alpha-equivalence”]

# Name Bindings

$$map\ f\ (map\ g\ arg) \longrightarrow map\ (\lambda x.\ f\ (g\ x))\ arg \qquad \text{(map-fusion)}$$

- Fresh name on every rewrite: inefficient
  - cannot find optimised 2D convolution[1]: out of memory after multiple minutes ✗
- DeBruijn indices: significant improvement
  - finds optimised 2D convolution[1] in 2s ✓

[1] *sequence of 13 rewrite rules, not counting $\alpha$ and $\beta$ reductions*

# Avoiding Name Bindings using Combinators

$$f\,(g\,x) \longrightarrow (f \circ g)\;x \qquad (\circ\text{-intro})$$

$$map\,f \circ map\,g \longrightarrow map\,(f \circ g) \qquad (\text{map-fusion}_2)$$

$$map\,(f \circ g) \longrightarrow map\,f \circ map\,g \qquad (\text{map-fission}_2)$$

instead of

$$map\,f\,(map\,g\;arg) \longrightarrow map\,(\lambda x.\,f(g(x)))\;arg \qquad (\text{map-fusion})$$

$$map\,(\lambda x.\,f\,g x) \longrightarrow \lambda y.\;map\,f\,(map\,(\lambda x.\,g x))\;y \qquad \text{if } x \text{ not free in } f \quad (\text{map-fission})$$

# Avoiding Name Bindings using Combinators

$$f\,(g\;x) \longrightarrow (f \circ g)\;x \qquad (\circ\text{-intro})$$

$$\mathit{map}\,f \circ \mathit{map}\;g \longrightarrow \mathit{map}\;(f \circ g) \qquad (\text{map-fusion}_2)$$

$$\mathit{map}\;(f \circ g) \longrightarrow \mathit{map}\,f \circ \mathit{map}\;g \qquad (\text{map-fission}_2)$$

instead of

$$\mathit{map}\,f\,(\mathit{map}\;g\;\mathit{arg}) \longrightarrow \mathit{map}\;(\lambda x.\,f(g(x)))\;\mathit{arg} \qquad (\text{map-fusion})$$

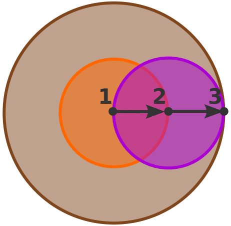
$$\mathit{map}\;(\lambda x.\,f\,gx) \longrightarrow \lambda y.\;\mathit{map}\,f\,(\mathit{map}\;(\lambda x.\,gx))\;y \qquad \text{if } x \text{ not free in } f \quad (\text{map-fission})$$

enables re-ordering 4D and tiling 2D loops in 30s ✓ *(with additional tricks)*

# Future Work

- We still cannot optimise large real-world Rise programs

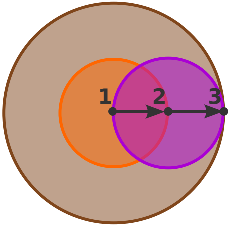
  *matrix multiplication, corner detection, ..*
- We want to explore many possibilities, but not all of them
- I am interested in focusing the search by:
  - deleting or filtering programs
  - enforcing normal forms
  - controlling optimisations using rewriting strategies
  - leveraging heuristics to prioritize promising directions



# Future Work

- We still cannot optimise large real-world Rise programs

  *matrix multiplication, corner detection, ..*
- We want to explore many possibilities, but not all of them
- I am interested in focusing the search by:
  - deleting or filtering programs
  - enforcing normal forms
  - controlling optimisations using rewriting strategies
  - leveraging heuristics to prioritize promising directions



---

✉ thomas.koehler@thok.eu
🌎 thok.eu

Thanks!

🌎 rise-lang.org
🌎 elevate-lang.org

# Adding Types

Consider $(\lambda x.\ x)\ (0 : int)$ and $(\lambda x.\ x)\ (0.0 : float)$

- Keeping types polymorphic enables more sharing:

$$\lambda x.\ x : \forall t.\ t \longrightarrow t$$

- Instantiating types enables more precise type-based rewriting:

$$\lambda x.\ x : \mathrm{int} \longrightarrow \mathrm{int}$$
$$\lambda x.\ x : \mathrm{float} \longrightarrow \mathrm{float}$$

Trade-off between amount of sharing and amount of information