Sketch-Guided Program Optimization

Thomas Kœhler   Phil Trinder   Michel Steuwer

University of Glasgow   The University of Edinburgh

TUM, Munich — July 2022

A Program Optimization Scenario

- imagine a performance engineer
- their task is to implement a high-performance matrix multiplication for a CPU
- they decide to do this by handwritting C code

Optimizing Matrix Multiplication in C

```c
for (int im = 0; im < m; im++) {
  for (int in = 0; in < n; in++) {
    float acc = 0.0f;
    for (int ik = 0; ik < k; ik++) {
      acc += a[ik + (k * im)] * b[in + (n * ik)];
    }

    output[in + (n * im)] = acc;
  }
}
```

Optimized program on the right:

- \+ 110× faster runtime

  Intel i5-4670K CPU

- \- 6× more lines, more complex code

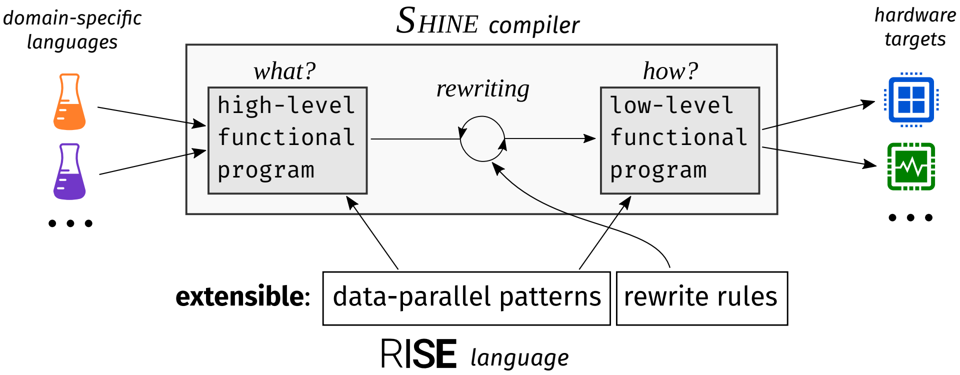
  threads, SIMD, indexing

```c
float aT[n * k];
#pragma omp parallel for
for (int in = 0; in < (n / 32); in = 1 + in) {
  for (int ik = 0; ik < k; ik = 1 + ik) {
    #pragma omp simd
    for (int jn = 0; jn < 32; jn = 1 + jn) {
      aT[(ik + ((32 * in) * k)) + (jn * k)] = a[(jn + (32 * in)) + (ik * n)];
    }
  }
}
#pragma omp parallel for
for (int im = 0; im < (m / 32); im = 1 + im) {
  for (int in = 0; in < (n / 32); in = 1 + in) {
    float tmp1[1024];
    for (int jm = 0; jm < 32; jm = 1 + jm) {
      for (int jn = 0; jn < 32; jn = 1 + jn) {
        tmp1[jn + (32 * jm)] = 0.0f;
      }
    }
    for (int ik = 0; ik < (k / 4); ik = 1 + ik) {
      for (int jm = 0; jm < 32; jm = 1 + jm) {
        float tmp2[32];
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] = tmp1[jn + (32 * jm)];
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((4 * ik) + ((32 * im) * k)) + (jm * k)] * aT[((4 * ik) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((1 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((1 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((2 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((2 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        #pragma omp simd
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp2[jn] += (a[((3 + (4 * ik)) + ((32 * im) * k)) + (jm * k)] *
            aT[((3 + (4 * ik)) + ((32 * in) * k)) + (jn * k)]);
        }
        for (int jn = 0; jn < 32; jn = 1 + jn) {
          tmp1[jn + (32 * jm)] = tmp2[jn];
        }
      }
    }
    for (int jm = 0; jm < 32; jm = 1 + jm) {
      for (int jn = 0; jn < 32; jn = 1 + jn) {
        output[((jn + ((32 * im) * n)) + (32 * in)) + (jm * n)] = tmp1[jn + (32 * jm)];
      }
    }
  }
}
```

Great performance, but time consuming and error-prone

How can we automate the optimization process?

Optimization via Term Rewriting



+ convenient, hardware agnostic programming
+ high-performance code generation
+ extensible set of abstractions and optimizations

Matrix Multiplication in RISE

High-level RISE program:

```
def mm a b =
  map (λaRow.                    | for aRow in a:
    map (λbCol.                  |   for bCol in transpose(b):
      dot aRow bCol)             |     ... = dot(aRow, bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0                     | for (x, y) in zip(xs, ys):
    (map (λ(x, y). x × y)        |   acc += x × y
      (zip xs ys))
```

Matrix Multiplication in RISE

High-level RISE program:

```
def mm a b =
  map (λaRow.                    | for aRow in a:
    map (λbCol.                  |   for bCol in transpose(b):
      dot aRow bCol)             |     ... = dot(aRow, bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0                     | for (x, y) in zip(xs, ys):
    (map (λ(x, y). x × y)        |   acc += x × y
      (zip xs ys))
```

RISE is a functional array language designed for optimization via term rewriting

Example RISE Rewrite Rules

split-join:

```
   map f x                 | for m:
                           |  ... = f(...)
     ↦
join
  (map                     | for m / n:
    (map f)                |  for n:
    (split n x))           |   ... = f(...)
```

transpose-around-map-map:

```
   map                     | for m:
     (map f) x             |  for n:
                           |   ... = f(...)
       ↦
transpose
  (map                     | for n:
    (map f)                |  for m:
    (transpose x))         |   ... = f(...)
```

Matrix Multiplication Blocking in RISE

```
map (λaRow.                | for m:
  map (λbCol.              |  for n:
    dot aRow bCol)         |   for k:
    (transpose b)) a       |    ...
```

$\mapsto^*$

```
join (map (map join) (map transpose
  map                                  | for m / 32:
    (map λx2.                          |  for n / 32:
      reduceSeq (λx3. λx4.             |   for k / 4:
        reduceSeq λx5. λx6.            |    for 4:
          map                          |     for 32:
            (map (λx7.                 |      for 32:
              (fst x7) + (fst (snd x7)) ×
                 (snd (snd x7)))
               (map (λx7. zip (fst x7) (snd x7))
                 (zip x5 x6)))
          (transpose (map transpose
          (snd (unzip (map unzip map (λx5.
            zip (fst x5) (snd x5))
            (zip x3 x4)))))))
        (generate (λx3. generate (λx4. 0)))
        transpose (map transpose x2))
    (map (map (map (map (split 4))))
      (map transpose
        (map (map (λx2. map (map (zip x2)
          (split 32 (transpose b)))))
            split 32 a))))))
```

How do we decide which rewrite rules to apply?

Rewriting Strategies

- programmers describe optimizations as compositions of rewrite rules
- MM blocking:

```
def blocking = ( baseline `;`
  tile(32,32)       `@` outermost(mapNest(2))   `;;`
  fissionReduceMap `@` outermost(appliedReduce)`;;`
  split(4)          `@` innermost(appliedReduce)`;;`
  reorder(List(1,2,5,6,3,4)))
```

+ empowers programmers to manually control the rewrite process
+ `tile`, `split`, `reorder` are not built-in but programmer-defined

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. “Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies”. In: ICFP (2020)

Rewriting Strategies

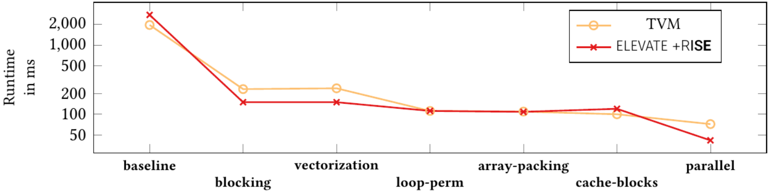
- programmers describe optimizations as compositions of rewrite rules
- MM blocking:

```
def blocking = ( baseline `;`
  tile(32,32)       `@` outermost(mapNest(2))   `;;`
  fissionReduceMap `@` outermost(appliedReduce)`;;`
  split(4)          `@` innermost(appliedReduce)`;;`
  reorder(List(1,2,5,6,3,4)))
```

- requires programmers to order all rewrite steps
- strategies are often restricted and complex to implement
- transformed program is hidden state that needs to be reasoned about

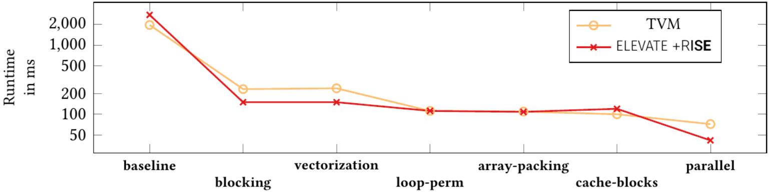
Rewriting Strategies

- Performance is on par with TVM for 7 different MM optimization goals:



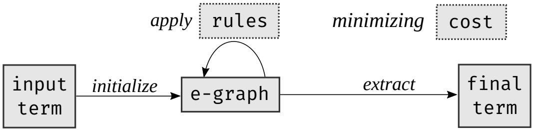
Rewriting Strategies

- Performance is on par with TVM for 7 different MM optimization goals:



Great performance, but requires manual rewrite ordering

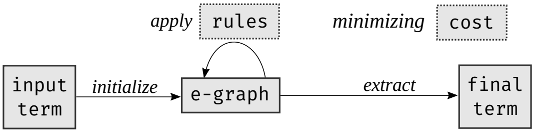
Equality Saturation



- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: fast and extensible equality saturation". In: POPL (2021)

Equality Saturation



- Optimize programs by efficiently exploring many possible rewrites
- Many successful applications sparked from the recent `egg` library

No manual rewrite ordering, but does not scale to MM optimizations in RISE

To overcome the limitations of rewriting strategies and equality saturation, we came up with *sketch-guided equality saturation*

Sketch-Guided Equality Saturation

Observation:

- The *shape* of the optimized program is often used to explain optimizations:

```
for m:
 for n:
  for k:
   ..
```

$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```

Sketch-Guided Equality Saturation

Observation:

- The *shape* of the optimized program is often used to explain optimizations:

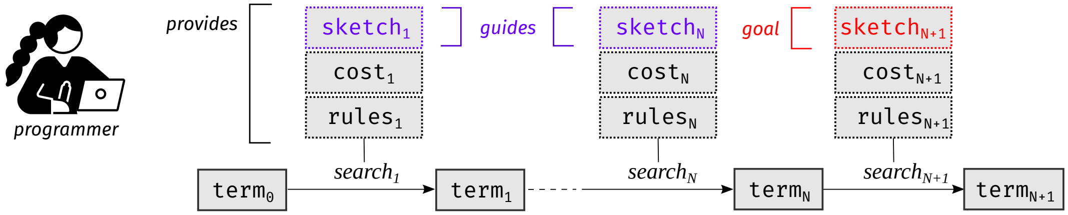
```
for m:
 for n:
  for k:
   ..
```

$\mapsto^*$

```
for m / 32:
 for n / 32:
  for k / 4:
   for 4:
    for 32:
     for 32:
      ..
```

Explanatory shapes can be formalized as sketches and used to guide rewriting

Sketch-Guided Equality Saturation



- Factors an unfeasible search into a sequence of feasible ones:
  1. Break long rewrite sequences
  2. Isolate explosive combinations of rewrite rules

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                   | for m:
 containsMap(n,                  |  for n:
  containsReduceSeq(k,           |   for k:
   containsAddMul)))             |     .. + .. × ..
```

- Abstractions defined in terms of smaller building blocks:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                    | for m:
 containsMap(n,                   |  for n:
  containsReduceSeq(k,            |   for k:
   containsAddMul)))              |     .. + .. × ..
```

- A sketch `s` is satisfied by a set of terms `R(s)`:

```
def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))

R(containsAddMul) = { R(app(app(+, ?), contains(×))) } ∪
  { F(t₁, .., tₙ) | ∃tᵢ ∈ R(containsAddMul) }
R(app(app(+, ?), contains(×))) = { app(app(+, t₁), t₂) | t₂ ∈ R(contains(×)) }
R(contains(×)) = { × } ∪ { F(t₁, .., tₙ) | ∃tᵢ ∈ R(contains(×)) }
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                    | for m:
 containsMap(n,                   |  for n:
  containsReduceSeq(k,            |   for k:
   containsAddMul)))              |     .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,               | for m / 32:
 containsMap(n / 32,              |  for n / 32:
  containsReduceSeq(k / 4,        |   for k / 4:
   containsReduceSeq(4,           |    for 4:
    containsMap(32,               |     for 32:
     containsMap(32,              |      for 32:
      containsAddMul))))))        |       .. + .. × ..
```

Sketches

- *Sketches* are program patterns that leave details unspecified

baseline sketch:

```
containsMap(m,                  | for m:
 containsMap(n,                 |  for n:
  containsReduceSeq(k,          |   for k:
   containsAddMul)))            |     .. + .. × ..
```

sketch guide:

how to split the loops before reordering them?

```
containsMap(m / 32,             | for m / 32:
 containsMap(32,                |  for 32:
  containsMap(n / 32,           |   for n / 32:
   containsMap(32,              |    for 32:
    containsReduceSeq(k / 4,    |     for k / 4:
     containsReduceSeq(4,       |      for 4:
      containsAddMul))))))      |        .. + .. × ..
```

blocking sketch:

```
containsMap(m / 32,             | for m / 32:
 containsMap(n / 32,            |  for n / 32:
  containsReduceSeq(k / 4,      |   for k / 4:
   containsReduceSeq(4,         |    for 4:
    containsMap(32,             |     for 32:
     containsMap(32,            |      for 32:
      containsAddMul))))))      |        .. + .. × ..
```

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

Sketch-guided equality saturation finds all 7 optimization goals

[2] Intel Xeon E5-2640 v2

[3] AMD Ryzen 5 PRO 2500U

Evaluation

- Equality Saturation without Sketch Guides[2]:

goal	found?	runtime	RAM
baseline	✓	0.5s	0.02 GB
blocking	✓	>1h	35 GB
+ 5 others	✗	>35mn	>60 GB

- Sketch-Guided Equality Saturation[3]:

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5s	0.02 GB
blocking	1	✓	7s	0.3 GB
+ 5 others	2-3	✓	≤7s	≤0.5 GB

582x

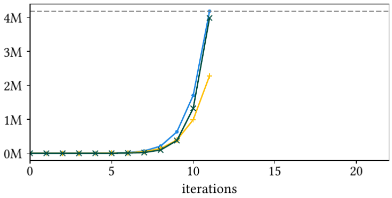
116x

[2] Intel Xeon E5-2640 v2

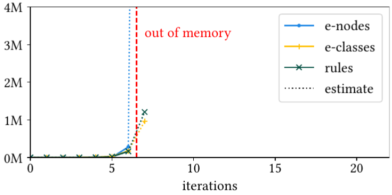
[3] AMD Ryzen 5 PRO 2500U

Evaluation

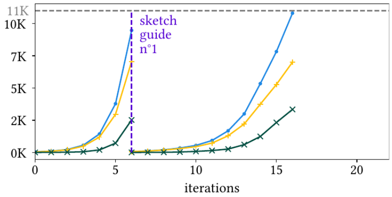
E-Graph Evolution



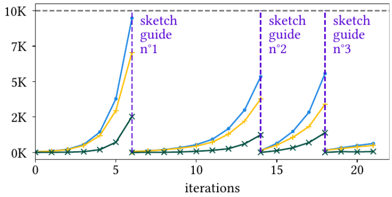
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗



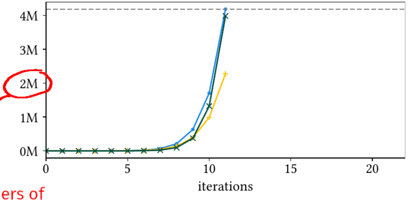
(c) sketch-guided, *blocking*, found: ✓



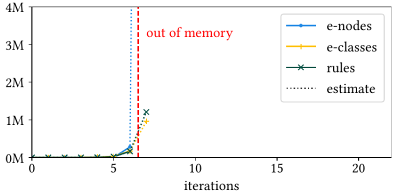
(d) sketch-guided, *parallel*, found: ✓

Evaluation

E-Graph Evolution

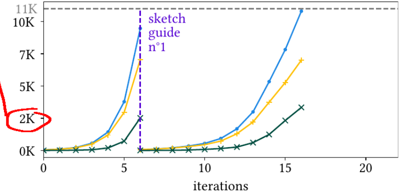


(a) unguided, *blocking*, found: ✓

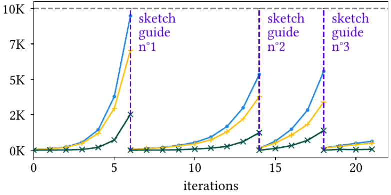


(b) unguided, *parallel*, found: ✗

three orders of magnitude



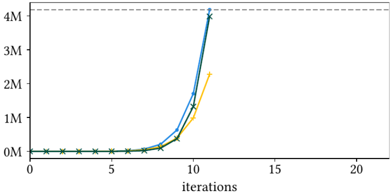
(c) sketch-guided, *blocking*, found: ✓



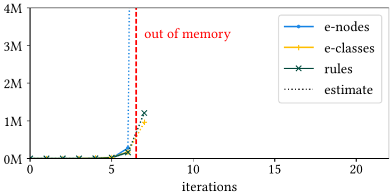
(d) sketch-guided, *parallel*, found: ✓

Evaluation

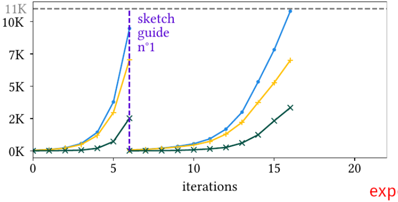
E-Graph Evolution



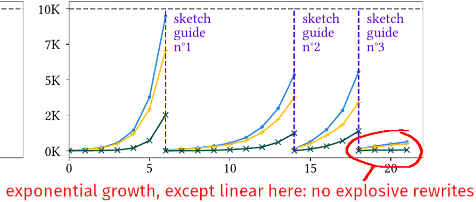
(a) unguided, *blocking*, found: ✓



(b) unguided, *parallel*, found: ✗



(c) sketch-guided, *blocking*, found: ✓



(d) sketch-guided, *parallel*, found: ✓

Evaluation

Sketches vs Full Program

all goals except *baseline*:

sketch guides	sketch goal	sketch sizes	program size
1-3	1	7-12	90-124

- each sketch corresponds to a logical transformation step
- sketches elide around 90% of the program
- sketches elide intricate details such as array reshaping patterns (e.g. `split`, `join`, `transpose`)

Conclusion

We propose:

- *sketches* to guide rewriting
- *sketch-guided equality saturation*, a novel, semi-automatic optimization technique

https://arxiv.org/abs/2111.13040

Conclusion

We propose:

- *sketches* to guide rewriting
- *sketch-guided equality saturation*, a novel, semi-automatic optimization technique

https://arxiv.org/abs/2111.13040

---

thomas.koehler@thok.eu
thok.eu

Thanks!

rise-lang.org
elevate-lang.org

Sketch Definition

$$S \;::=\; ? \;\mid\; F(S, .., S) \;\mid\; contains(S)$$

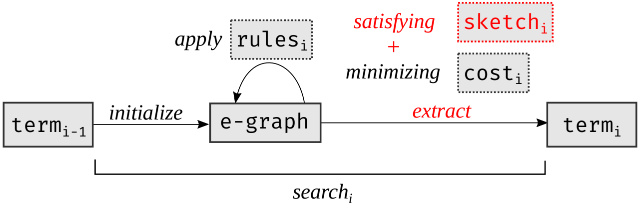
$$R(?) = T = \{F(t_1, .., t_n)\}$$
$$R(F(s_1, .., s_n)) = \{F(t_1, .., t_n) \mid t_i \in R(s_i)\}$$
$$R(contains(s)) = R(s) \cup \{F(t_1, .., t_n) \mid \exists t_i \in R(contains(s))\}$$

```
def containsMap(n: NatSketch, f: Sketch): Sketch =
  contains(app(map :: ?t → n.?dt → ?y, f))

def containsReduceSeq(n: NatSketch, f: Sketch): Sketch =
  contains(app(reduceSeq :: ?t → ?t → n.?dt → ?t, f))

def containsAddMul: Sketch =
  contains(app(app(+, ?), contains(×)))
```

Sketch-Satisfying Equality Saturation



- Terminates as soon as a program satisfying the sketch is found

Sketches vs Full Program

goal	sketch guides	sketch goal	sketch sizes	program size
blocking	*split*	$reorder_1$	7	90
vectorization	*split* + $reorder_1$	$lower_1$	7	124
loop-perm	*split* + $reorder_2$	$lower_2$	7	104
array-packing	*split* + $reorder_2$ + *store*	$lower_3$	7-12	121
cache-blocks	*split* + $reorder_2$ + *store*	$lower_4$	7-12	121
parallel	*split* + $reorder_2$ + *store*	$lower_5$	7-12	121

- each sketch corresponds to a logical transformation step
- sketches elide around 90% of the program
- intricate details such as array reshaping patterns are not specified (e.g. `split`, `join`, `transpose`)

Deciding How to Apply Rewrite Rules

Fully automated search?

e.g. heuristic search,
equality saturation, …

This is fully **automated**!

The search takes forever ..

Why is the performance so bad in this case?

I wish I could have control over the optimizations!

Manually written recipe?

e.g. Halide/TVM schedules,
Elevate strategies, …

I have full **control**!

This was more difficult to write than I expected!

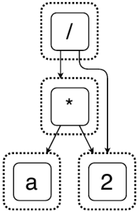
How can I generalize this recipe to other cases?

Guided search!

I can combine **control and automation**!

E-Graph Example

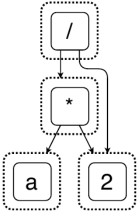
$$(a * 2)/2 \longrightarrow^* a$$



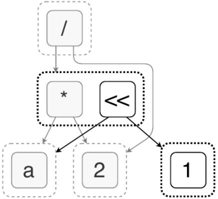
$(a * 2)/2$

E-Graph Example

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



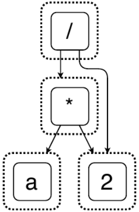
$(a * 2)/2$



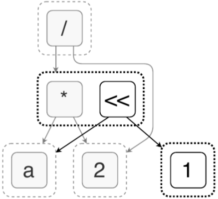
$x * 2 \longrightarrow x \ll 1$

E-Graph Example

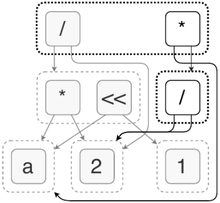
$$(a * 2)/2 \longrightarrow^{*} a$$



$(a * 2)/2$



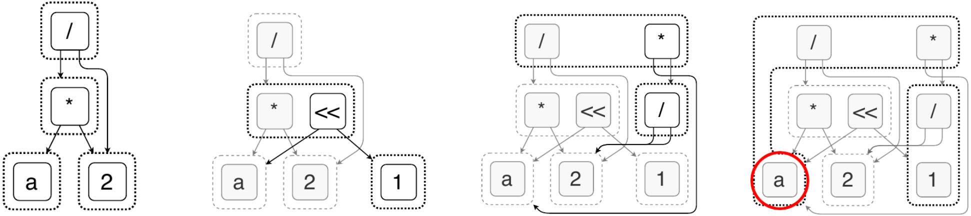
$x * 2 \longrightarrow x \ll 1$



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

E-Graph Example

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



$(a * 2)/2$

$x * 2 \longrightarrow x \ll 1$

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

$x/x \longrightarrow 1$

$x * 1 \longrightarrow x$

cost = term size