Guided Equality Saturation

Saturation d'Égalité Guidée

Thomas Kœhler Andrés Goens Siddharth Bhat Tobias Grosser
Phil Trinder Michel Steuwer

Journées AFADL / GDR GPL — Strasbourg, 5 Juin, 2024

Term Rewriting

Applying sets of rewrite rules to terms to **achieve diverse goals**:

- Optimizing programs in compilers (e.g. LLVM, GHC)
- Proving theorems in solvers (e.g. Z3) and proof assistants (e.g. Coq, Lean)

$$a + (b - a) \mapsto^* b \qquad \text{using} \begin{cases} x + (y - z) \leftrightarrow (x + y) - z \\ x + y \leftrightarrow y + x \\ x - x \mapsto 0 \\ x + 0 \mapsto x \end{cases}$$

- Complex transformations are obtained **by composition** of simple rewrite rules.
- Achieving a goal **requires deciding how to apply** rewrite rules.

Greedy Term Rewriting

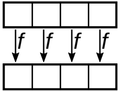
Example: Eliminating Intermediate Memory with Fusion

Rewrite Rules:

(1) map a ∘ map b ↦ map (a ∘ b)

uses less memory

map f:




```
alloc out
for i:
  out[i] = f(in[i])
```

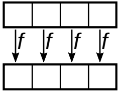
Greedy Term Rewriting

Example: Eliminating Intermediate Memory with Fusion

Rewrite Rules:

(1) $map\ a \circ map\ b \mapsto \underline{map\ (a \circ b)}$

uses less memory

map f: 


```
alloc out
for i:
  out[i] = f(in[i])
```

Program to Optimize:

$(map\ (map\ f)) \circ (map\ (map\ g))$

greedy rewriting: $\downarrow$ (1)

$map\ ((map\ f) \circ (map\ g))$

$\downarrow$ (1)

$map\ (map\ (f \circ g))$

The Limits of Greedy Term Rewriting

Example: Eliminating Intermediate Memory with Fusion

Rewrite Rules:

(1) map a ∘ map b ↦ map (a ∘ b)

uses less memory

Program to Optimize:

(map (map f)) ∘ (transpose ∘ (map (map g)))

$\not\downarrow$

local optimum,
cannot use less memory?

The Limits of Greedy Term Rewriting

Example: Eliminating Intermediate Memory with Fusion

Rewrite Rules:

$(1)\ \mathit{map}\ a \circ \mathit{map}\ b \mapsto \underbrace{\mathit{map}\ (a \circ b)}_{\text{uses less memory}}$

$(2)\ \mathit{transpose} \circ \mathit{map}\ (\mathit{map}\ a) \mapsto \mathit{map}\ (\mathit{map}\ a) \circ \mathit{transpose}$

$(3)\ a \circ (b \circ c) \mapsto (a \circ b) \circ c$

no effect on memory

Program to Optimize:

$(\mathit{map}\ (\mathit{map}\ f)) \circ (\mathit{transpose} \circ (\mathit{map}\ (\mathit{map}\ g)))$

not explored by greedy rewriting: $\big\downarrow$ (2); (3)

$((\mathit{map}\ (\mathit{map}\ f)) \circ (\mathit{map}\ (\mathit{map}\ g))) \circ \mathit{transpose}$

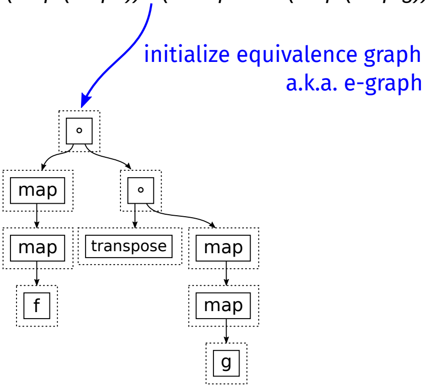
$\big\downarrow$ (1); (1)

$\underbrace{(\mathit{map}\ (\mathit{map}\ (f \circ g))) \circ \mathit{transpose}}_{\text{global optimum, uses less memory!}}$

Equality Saturation

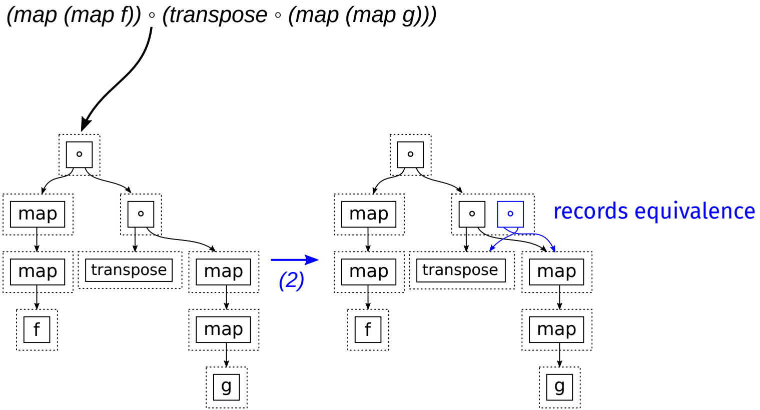
Example: Eliminating Intermediate Memory with Fusion

(map (map f)) ∘ (transpose ∘ (map (map g)))



Equality Saturation

Example: Eliminating Intermediate Memory with Fusion

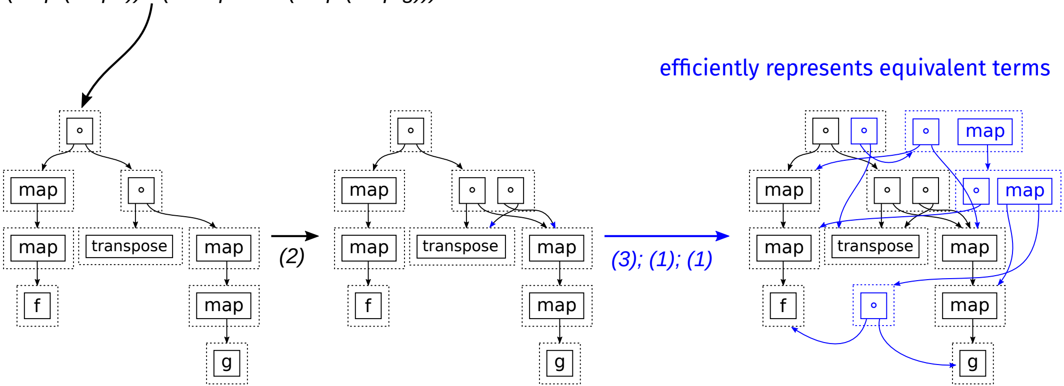


transpose ∘ *map (map g)*
=
map (map g) ∘ *transpose*

Equality Saturation

Example: Eliminating Intermediate Memory with Fusion

(map (map f)) ∘ (transpose ∘ (map (map g)))

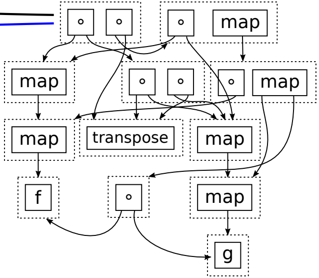


Equality Saturation

Example: Eliminating Intermediate Memory with Fusion

(map (map f)) ∘ (transpose ∘ (map (map g)))

extract term minimizing memory usage



(map (map (f ∘ g))) ∘ transpose

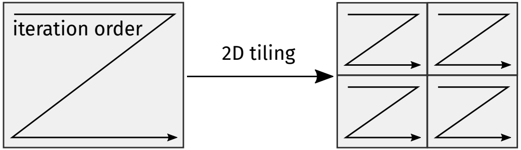
global optimum,
found by equality saturation [Tate et al. 2009; Willsey et al. 2021]

The Limits of Equality Saturation

Example: Improving Memory Access Patterns with Tiling

$(1),(2),(3) + (4)\ map\ n_1\ (map\ n_2\ f) \mapsto transpose \circ (map\ n_2\ (map\ n_1\ f)) \circ transpose$

$(5)\ map\ (n_1 \times n_2)\ f \mapsto join \circ (map\ n_1\ (map\ n_2\ f)) \circ (split\ n_2)$




```
for i:
  T[i]
```

(5) →

```
for j:
  for k:
    T[j * n + k]
```

(4) →

```
for k:
  for j:
    T[j * n + k]
```

The Limits of Equality Saturation

Example: Improving Memory Access Patterns with Tiling

(1),(2),(3) + (4) map n_1 (map n_2 f) ↦ transpose ∘ (map n_2 (map n_1 f)) ∘ transpose

(5) map ($n_1 \times n_2$) f ↦ join ∘ (map n_1 (map n_2 f)) ∘ (split n_2)

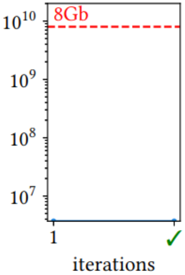
1D Tiling:

map (n_1 × 32) f

↓ easy!

join ∘ (map n_1 (map 32 f)) ∘ (split 32)

Memory Footprint (bytes):



1 iteration =
apply rules breadth-first

The Limits of Equality Saturation

Example: Improving Memory Access Patterns with Tiling

(1),(2),(3) + (4) $map\ n_1\ (map\ n_2\ f) \mapsto transpose \circ (map\ n_2\ (map\ n_1\ f)) \circ transpose$

(5) $map\ (n_1 \times n_2)\ f \mapsto join \circ (map\ n_1\ (map\ n_2\ f)) \circ (split\ n_2)$

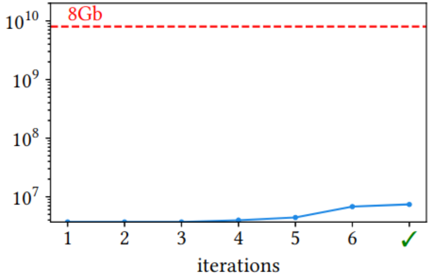
2D Tiling:

$map\ (n_1 \times 32)\ (map\ (n_2 \times 32)\ f)$

↓ more challenging …

$join \circ (map\ n_1\ (map\ 32\ join)) \circ$
$(map\ n_1\ transpose) \circ$
$(map\ n_1\ (map\ n_2\ (map\ 32\ (map\ 32\ f)))) \circ$
$(map\ n_1\ transpose) \circ$
$(map\ n_1\ (map\ 32\ (split\ 32))) \circ (split\ 32)$

Memory Footprint (bytes):



The Limits of Equality Saturation

Example: Improving Memory Access Patterns with Tiling

(1),(2),(3) + (4) $map\ n_1\ (map\ n_2\ f) \mapsto transpose \circ (map\ n_2\ (map\ n_1\ f)) \circ transpose$

(5) $map\ (n_1 \times n_2)\ f \mapsto join \circ (map\ n_1\ (map\ n_2\ f)) \circ (split\ n_2)$

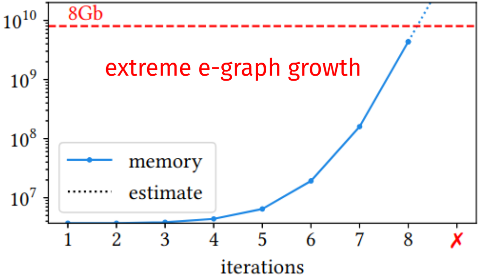
3D Tiling:

$map\ (n_1 \times 32)\ (map\ (n_2 \times 32)\ (map\ (n_3 \times 32)\ f))$

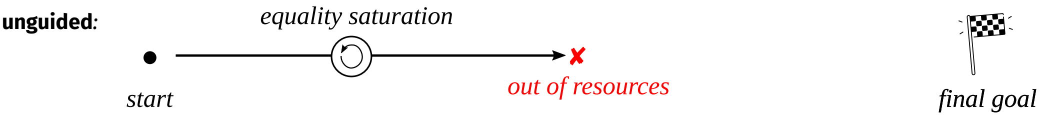
unreachable with 8 Gb!

$(map\ (n_1 \times 32)\ (map\ (n_2 \times 32)\ join) \circ join) \circ join \circ$
$(map\ n_1\ transpose \circ (map\ n_2\ (map\ 32\ transpose) \circ transpose)) \circ$
$(map\ n_1\ (map\ n_2\ (map\ n_3\ (map\ 32\ (map\ 32\ (map\ 32\ f)))))) \circ$
$(map\ n_1\ (map\ n_2\ transpose)) \circ$
$(map\ n_1\ (map\ n_2\ (map\ 32\ transpose)) \circ transpose) \circ$
$(split\ 32) \circ (map\ (n_1 \times 32)\ (map\ n_2\ (map\ 32\ (split\ 32))) \circ (split\ 32))$

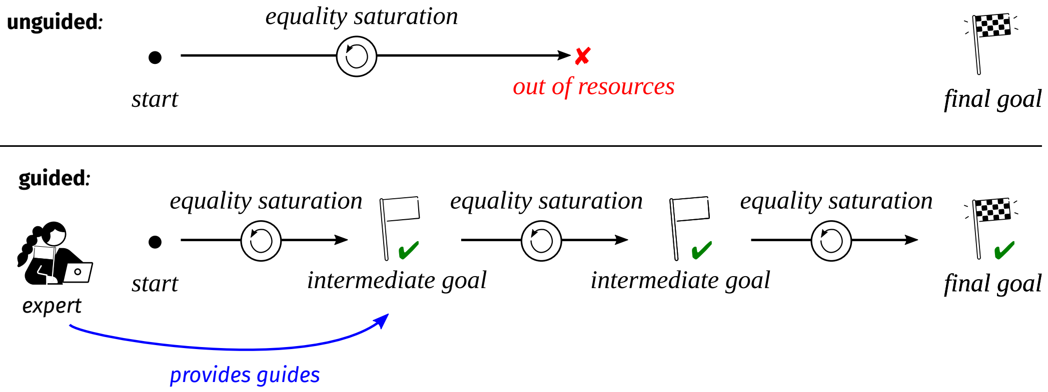
Memory Footprint (bytes):



Guided Equality Saturation



Guided Equality Saturation



Guided Equality Saturation

Example: Improving Memory Access Patterns with Tiling (3D)

$map\ (n_1 \times 32)\ (map\ (n_2 \times 32)\ (map\ (n_3 \times 32)\ f))$

```
for i:
  for j:
    for k:
      ...
```

1. split the loops

guide provides insight

$(map\ (n1 \times 32)\ (map\ (n2 \times 32)\ join) \circ join) \circ join\ \circ$
$(map\ n1\ (map\ 32\ (map\ n2\ (map\ 32\ (map\ n3\ (map\ 32\ f)))))) \circ$
$(split\ 32) \circ (map\ (n1 \times 32)\ (map\ n2\ (map\ 32\ (split\ 32))) \circ (split\ 32))$

```
for i1: for i2:
  for j1: for j2:
    for k1: for k2:
      ...
```

2. reorder them

$(map\ (n_1 \times 32)\ (map\ (n_2 \times 32)\ join) \circ join) \circ join\ \circ$
$(map\ n_1\ transpose \circ (map\ n_2\ (map\ 32\ transpose) \circ transpose)) \circ$
$(map\ n_1\ (map\ n_2\ (map\ n_3\ (map\ 32\ (map\ 32\ (map\ 32\ f)))))) \circ$
$(map\ n_1\ (map\ n_2\ transpose)) \circ$
$(map\ n_1\ (map\ n_2\ (map\ 32\ transpose)) \circ transpose) \circ$
$(split\ 32) \circ (map\ (n_1 \times 32)\ (map\ n_2\ (map\ 32\ (split\ 32))) \circ (split\ 32))$

```
for i1: for j1: for k1:
  for i2: for j2: for k2:
    ...
```

Guided Equality Saturation

Example: Improving Memory Access Patterns with Tiling (3D)

map $(n_1 \times 32)$ *(map* $(n_2 \times 32)$ *(map* $(n_3 \times 32)$ *f))*

```
for i:
  for j:
    for k:
      ...
```

1. split the loops

guide provides insight

(contains
(map n1 (map 32 (map n2 (map 32 (map n3 (map 32 f))))))
)

more or less precise **sketch**

```
for i1: for i2:
  for j1: for j2:
    for k1: for k2:
      ...
```

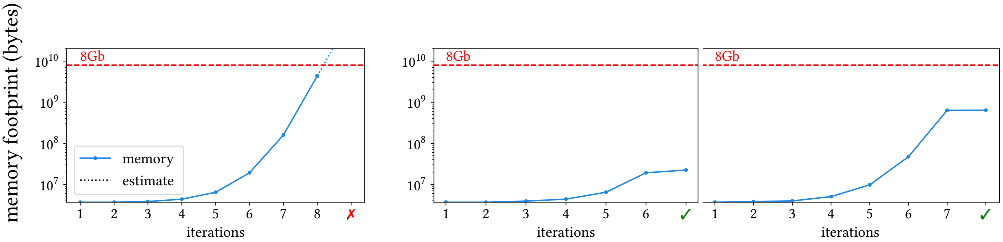
2. reorder them

(map $(n_1 \times 32)$ *(map* $(n_2 \times 32)$ *join)* ∘ *join)* ∘ *join* ∘
(map n_1 *transpose* ∘ *(map* n_2 (map 32 transpose) ∘ transpose)) ∘
(map n_1 *(map* n_2 *(map* n_3 *(map 32 (map 32 (map 32 f))))))* ∘
(map n_1 *(map* n_2 *transpose))* ∘
(map n_1 *(map* n_2 *(map 32 transpose))* ∘ *transpose)* ∘
(split 32) ∘ *(map* $(n_1 \times 32)$ *(map* n_2 *(map 32 (split 32)))* ∘ *(split 32))*

```
for i1: for j1: for k1:
  for i2: for j2: for k2:
    ...
```

Guided Equality Saturation

Example: Improving Memory Access Patterns with Tiling (3D)



(a) Equality saturation (found: ✗)

(b) **Guided** equality saturation (found: ✓)

- A single guide makes 3D Tiling reachable with 8Gb!

Case Study: Program Optimization

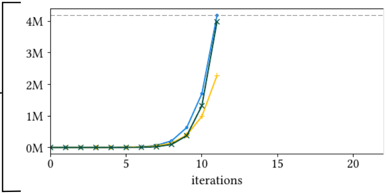
- We reproduced Matrix Multiplication optimizations from TVM:
  https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html
  - transform loops *blocking, permutation, unrolling*
  - change data layout
  - add parallelism *vectorization, multi-threading*
- Prior work performs them by manually composing rewrite rules *[ICFP 2020; CACM 2023]*

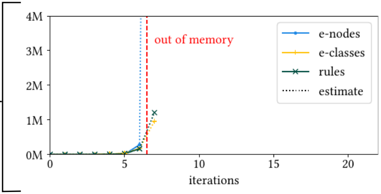
Case Study: Program Optimization

Unguided Runtime and Memory Consumption

goal	found?	runtime	RAM
baseline	✓	0.5 s	0.02 GB
blocking	✓	>1 h	35 GB
vectorization	✗	>1 h	>60 GB
loop-perm	✗	>1 h	>60 GB
array-packing	✗	35 m	>60 GB
cache-blocks	✗	35 m	>60 GB
parallel	✗	35 m	>60 GB

5 goals are too hard to find with unguided equality saturation





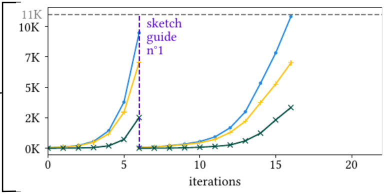
Case Study: Program Optimization

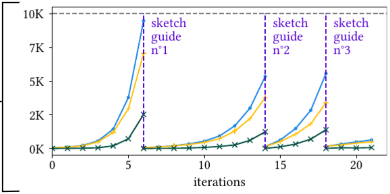
Guided Runtime and Memory Consumption

582x faster 116x less

goal	sketch guides	found?	runtime	RAM
baseline	0	✓	0.5 s	0.02 GB
blocking	1	✓	7 s	0.3 GB
vectorization	2	✓	7 s	0.4 GB
loop-perm	2	✓	4 s	0.3 GB
array-packing	3	✓	5 s	0.4 GB
cache-blocks	3	✓	5 s	0.5 GB
parallel	3	✓	5 s	0.4 GB

All found with up to 3 guides,
eliding 90% of the complete program





Case Study: Theorem Proving

- We implemented a **ges** tactic for the Lean theorem prover:

```
g⁻¹⁻¹ = g⁻¹⁻¹ * (g⁻¹ * g)
  _ = g
```

key reasoning step

- Steps and details are omitted:

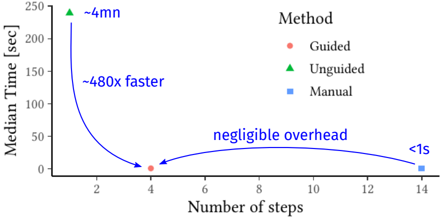
$$(g^{-1})^{-1} \xrightarrow{\text{mul. one}} (g^{-1})^{-1} \cdot 1 \xrightarrow{\text{mul. inverse}} (g^{-1})^{-1} \cdot (g^{-1} \cdot g)$$

$$\xrightarrow{\text{mul. assoc.}} ((g^{-1})^{-1} \cdot g^{-1}) \cdot g \xrightarrow{\text{mul. inverse}} 1 \cdot g \xrightarrow{\text{mul. one}} g$$

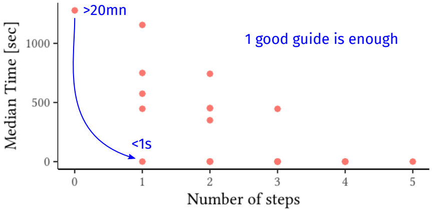
Case Study: Theorem Proving

Proving Theorems on Rings of Characteristic 2*

$$(x+y)^2 = x^2 + y^2$$



$$(x+y)^3 = x^3 + x \cdot y^2 + x^2 \cdot y + y$$



$^*1 + 1 = 0, x + x = 0$

Conclusion

- **Guided Equality Saturation** offers an effective trade-off between manual and automated rewriting
- For program optimization, guides resemble explanatory code snippets
- For theorem proving, guides resemble key reasoning steps from textbooks
- More details in our paper, supplementary material and open-source code!

Conclusion

- **Guided Equality Saturation** offers an effective trade-off between manual and automated rewriting
- For program optimization, guides resemble explanatory code snippets
- For theorem proving, guides resemble key reasoning steps from textbooks
- More details in our paper, supplementary material and open-source code!

Future Work:

- increasing the expressivity of sketches *(e.g. named holes, intersection, negation)*
- proving more challenging theorems from textbooks *(e.g. requiring conditional rewrites)*
- generalizing to imperative code optimization
- improving user experience and providing feedback
- more combinations of human intuition with automation!

Conclusion

- **Guided Equality Saturation** offers an effective trade-off between manual and automated rewriting
- For program optimization, guides resemble explanatory code snippets
- For theorem proving, guides resemble key reasoning steps from textbooks
- More details in our paper, supplementary material and open-source code!

Future Work:

- increasing the expressivity of sketches *(e.g. named holes, intersection, negation)*
- proving more challenging theorems from textbooks *(e.g. requiring conditional rewrites)*
- generalizing to imperative code optimization
- improving user experience and providing feedback
- more combinations of human intuition with automation!

Thanks! thok.eu

Sketch Definition

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

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

```
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(×)))
```

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)))               |     .. + .. × ..
```

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))))))         |        .. + .. × ..
```

Rewritten Language

- Rewritten language: RISE, a functional array language

Matrix Multiplication in RISE:

```
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))
```

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`)

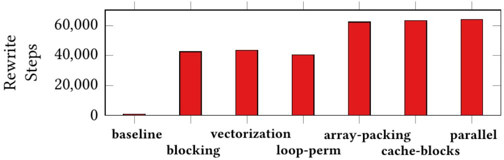
Difficulty 1. Long Rewrite Sequences

```
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))))))
```

Prior work *(not shortest path)*:



Difficulty 2. Explosive Combinations of Rewrite Rules

Two example rules that quickly generate many possibilities:

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(...)
```

Handwritten Matrix Multiplication

```
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;
  }
}
```

```
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)];
      }
    }
  }
}
```

Optimised program on the right:

+ 110× faster runtime

  Intel i5-4670K CPU

- 6× more lines of code where things can go wrong

  threads, SIMD, index computations

- hardware specific (not portable)