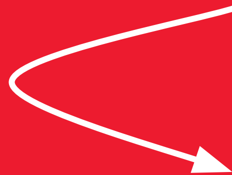
Foresight — **Parallel and Customizable Equality Saturation**

Use cases

- Superoptimize tensor graphs (MLSys '21)
- Recognize hidden idioms (CGO '24)
- Share hardware on FPGAs (CGO '26)

Challenges

- Scalability
- Flexibility

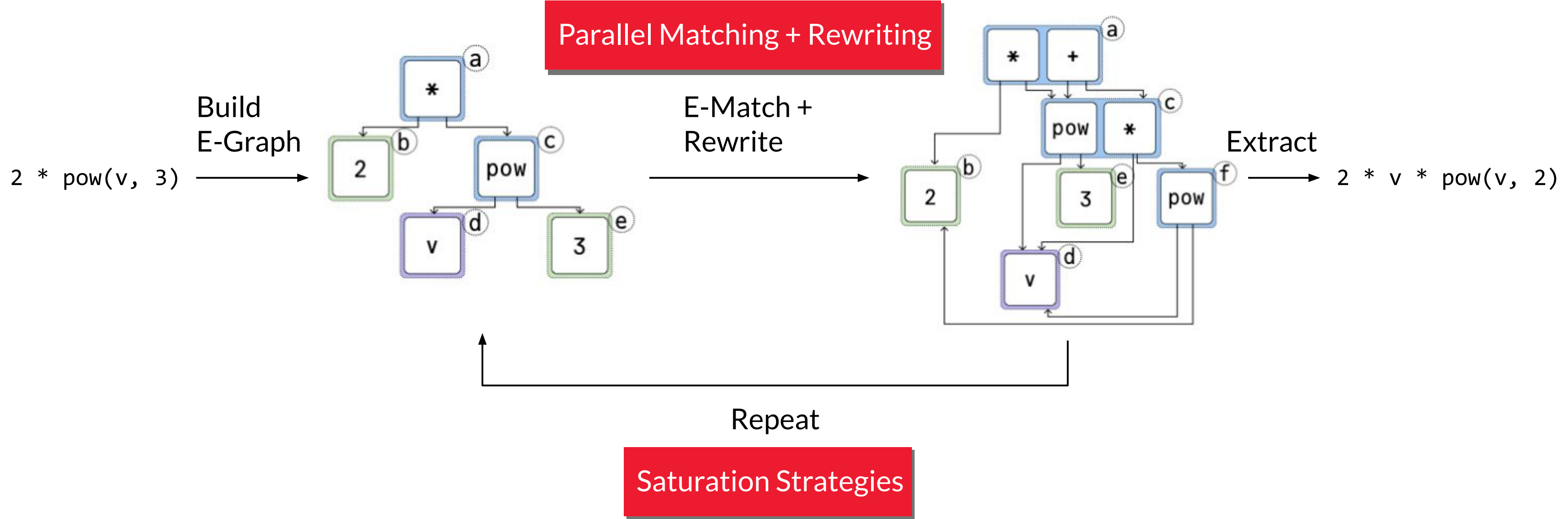


Foresight: Scala EqSat library rooted in **Parallelism + Customizability**

McGill

Jonathan Van der Cruysse, Abd-El-Aziz Zayed,
Mai Jacob Peng, Christophe Dubach
McGill University – CC '26 – Saturday January 31, 2026

Equality Saturation

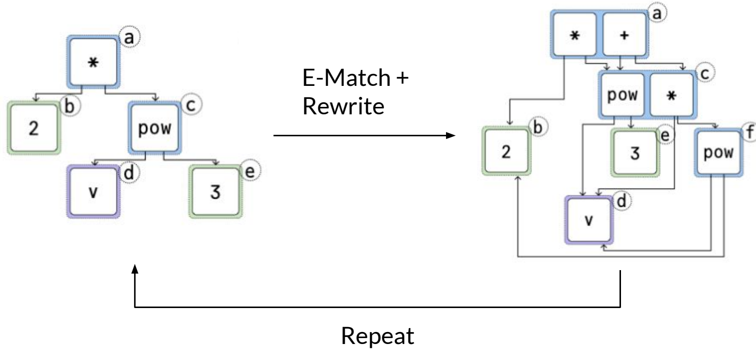


Saturation Loops and Library APIs

```
def saturate(egraph, rules):
  while True:
    egraph’ = rewrite(egraph, rules)
    if egraph’ == egraph:
      return egraph’

    egraph = egraph’
```

Vanilla saturation:

```
saturate(egraph, rules)
```



What if full saturation is too expensive?

Saturation Loops and Library APIs

```
def saturate(egraph, rules, condition, scheduler):
  i = 0
  while True:
    egraph’ = rewrite(
      egraph, scheduler(rules, i))

    if egraph’ == egraph
      or condition(egraph’):

      return egraph’

    egraph = egraph’
    i += 1
```

Timeout:

```
saturate(egraph, rules, condition = TIMEOUT(60s))
```

Exponential backoff:

```
saturate(egraph, rules, scheduler = BACKOFF(n, c))
```

Rebasing? Multi-phase saturation? E-graph segmentation?
→ No solution out of the box
→ Build your own on top of `saturate`

Rebasing, Multi-Phase Saturation

```
def my_saturate(
    egraph, rules1, rules2, iterations,
    timeout1, timeout2, n1, c1, n2, c2):

  for i in 0..iterations:
    egraph' = saturate(
      egraph,
      rules1,
      condition = TIMEOUT(timeout1),
      scheduler = BACKOFF(n1, c1))
    egraph'' = saturate(
      egraph',
      rules2,
      condition = TIMEOUT(timeout2),
      scheduler = BACKOFF(n2, c2))

    expression = extract(egraph'')
    egraph = egraph_from(expression)

  return egraph
```

Saturation Strategies

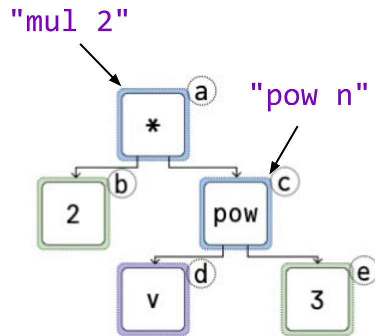
```
def phase(rules, timeout, n, c):
  return BackOffRuleApplication(rules, n, c)
  .withTimeout(timeout)
  .repeatUntilStable
strategy =
  phase(rules1, timeout1, n1, c1)
  .thenApply(phase(rules2, timeout2, n2, c2))
  .thenRebase(extract)
  .withIterationLimit(iterations)
  .repeatUntilStable
```

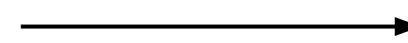
```
egraph’ = strategy(egraph)
```

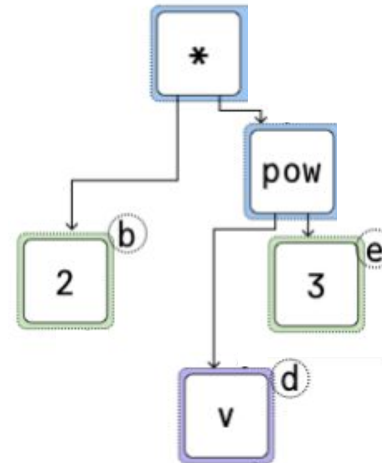
E-Matching and Rewriting

```
"pow n": pow(x, n) ⇒ x * pow(x, n - 1)
"mul 2": 2 * y     ⇒ y + y
```

```
"pow n" @ c: pow(d, 3) ⇒ d * pow(d, 2)
"mul 2" @ a: 2 * c     ⇒ c + c
```

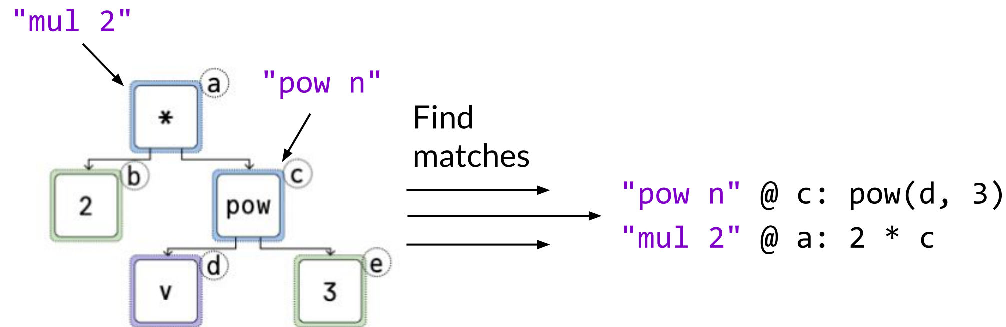






Parallel E-Matching

```
"pow n": pow(x, n) ⇒ x * pow(x, n - 1)
"mul 2": 2 * y     ⇒ y + y
```

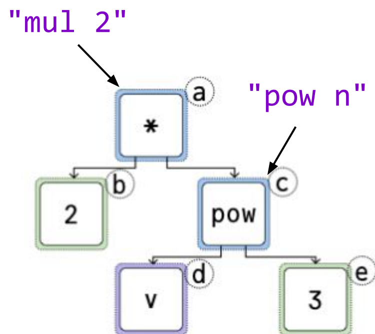


Thread-Safe E-Graph

Parallel Rewriting

```
"pow n": pow(x, n) ⇒ x * pow(x, n - 1)
"mul 2": 2 * y     ⇒ y + y
```

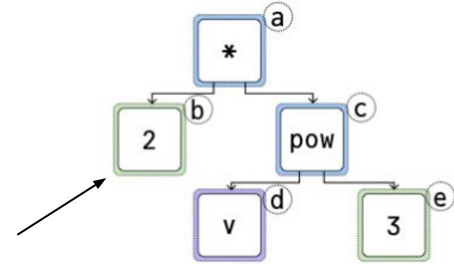
```
"pow n" @ c: pow(d, 3) ⇒ d * pow(d, 2)
"mul 2" @ a: 2 * c     ⇒ c + c
```



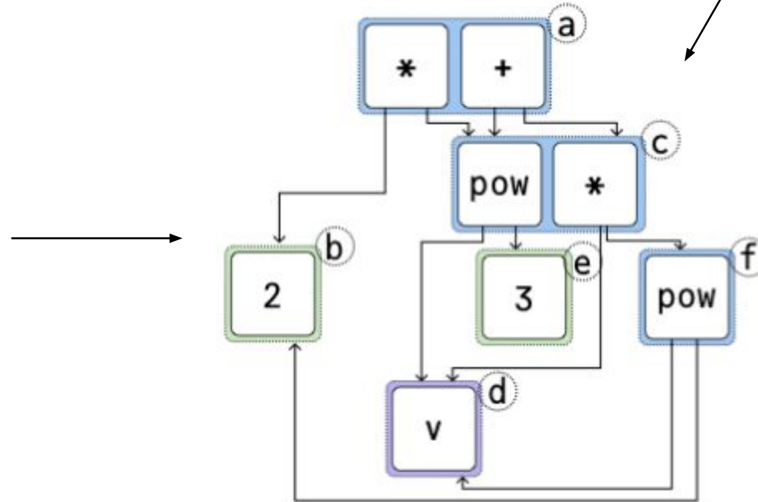

```
// first match
%0 = add (2)
%1 = add (pow d, %0)
%2 = add (mul d, %1)
union %2, c

// second match
%3 = add (c, c)
union %3, a
```

SSA instructions

Parallel Rewriting




```
// first match
%0 = add (2)
%1 = add (pow d, %0)
%2 = add (mul d, %1)
union %2, b
%2 = add (mul d, %1)
// second match
%3 = add (e, b)
union %3, a
union %2, c
```



SSA instructions

Parallel E-Matching

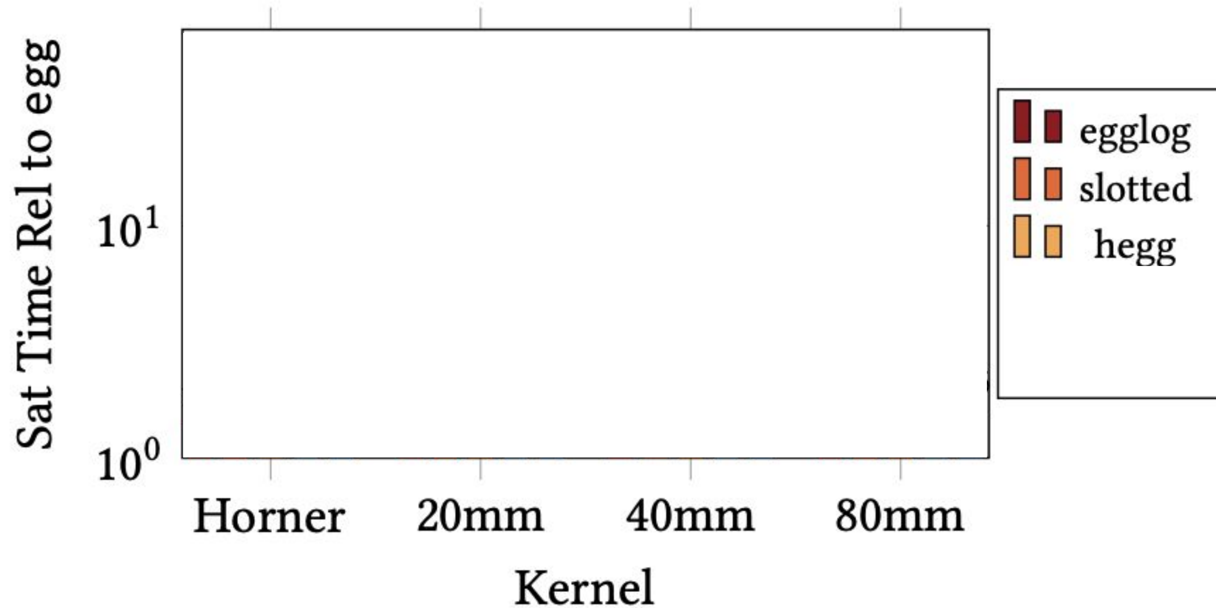
```
egraph' = strategy(egraph, ParallelMap.parallel)
                           ParallelMap.sequential
                           ParallelMap.fixedThreadParallel(n)
```

Evaluation

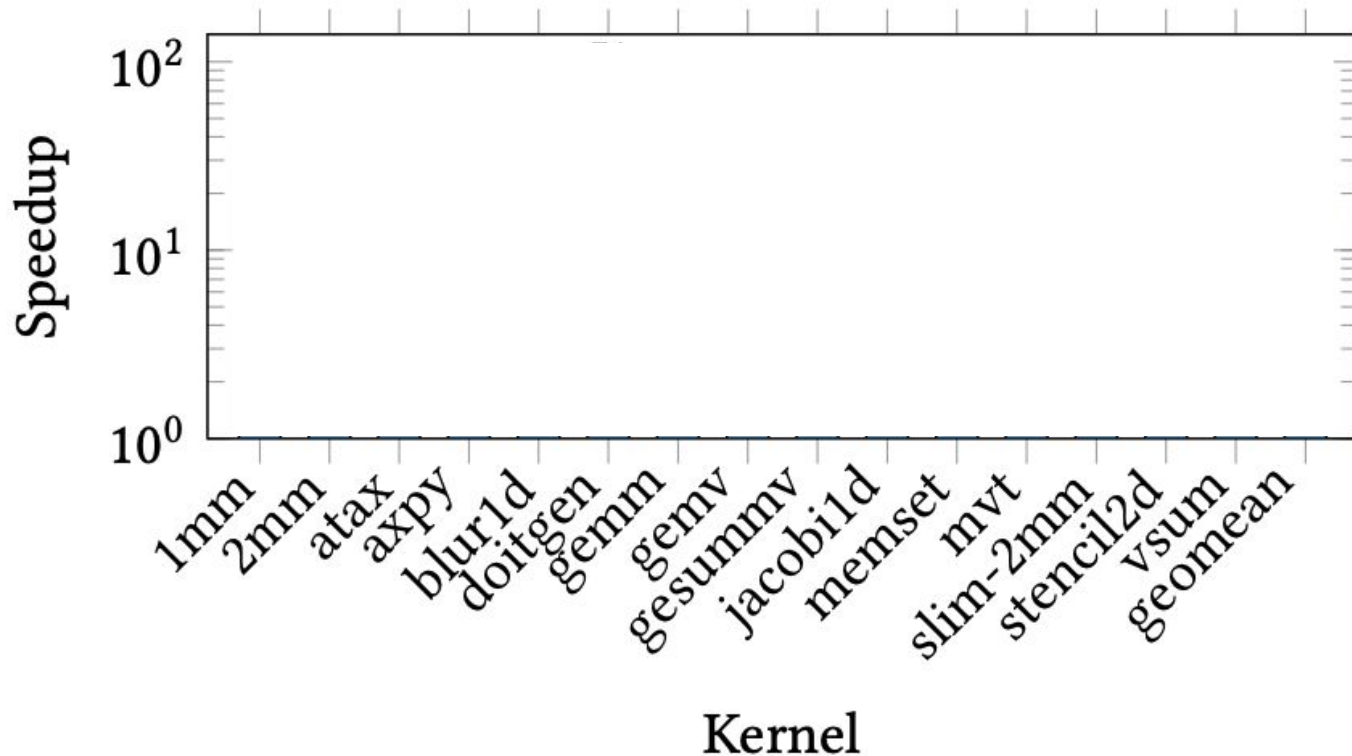
Case studies

- **Microbenchmarks: Horner and matmul chains**
- **Idiom recognition reimplementation**
- + more in the paper

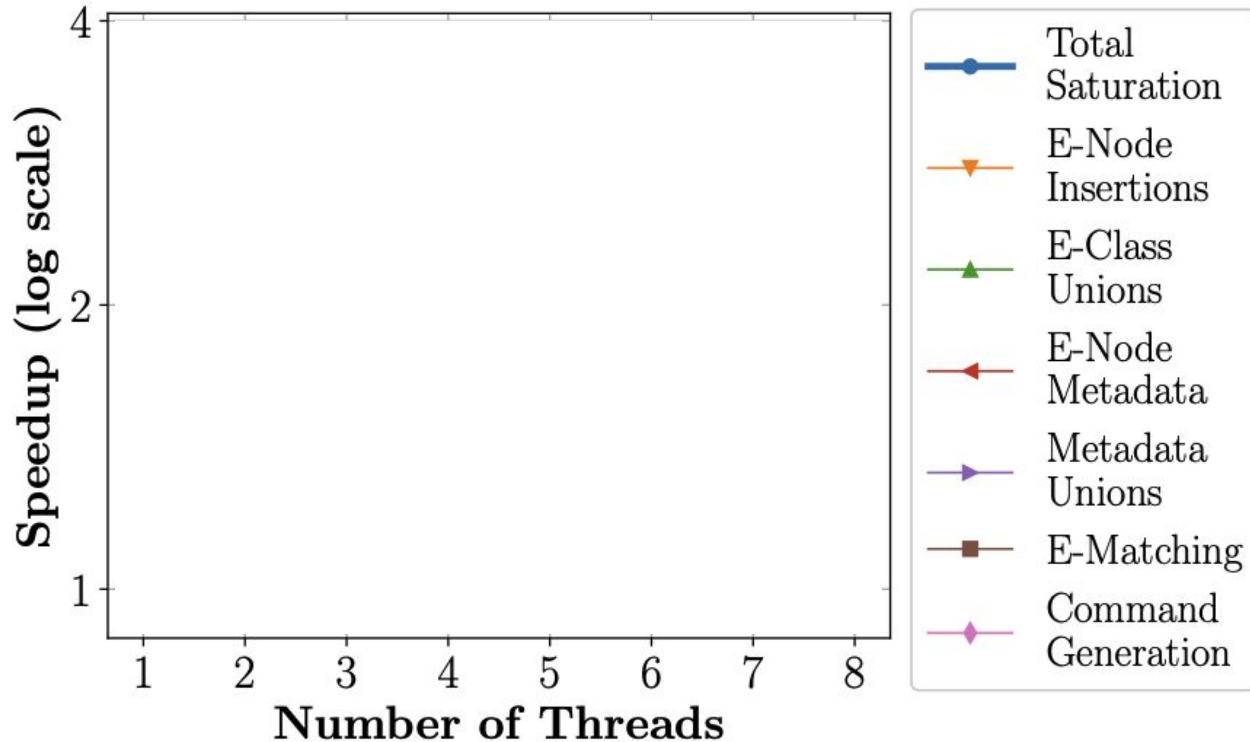
Speedups: Horner, matmul chains



Speedups: LIAR idiom recognition



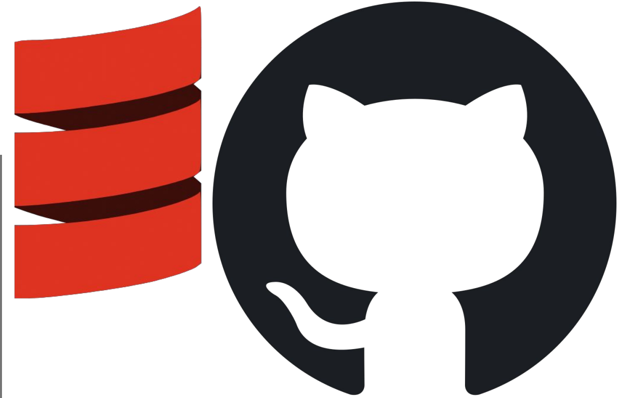
Parallel Scaling: LIAR (stencil2d)



Conclusion

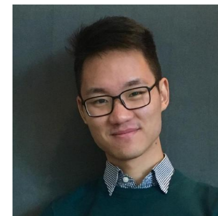
Three ideas

- **Saturation strategies**
- **Parallel e-matching and rewriting**
- Generalized metadata



github.com/jonathanvdc/foresight

Foresight – Parallel and Customizable Equality Saturation



Jonathan Van der Cruysse
McGill University
jonathan.vandercruysse@mail.mcgill.ca

Abd-El-Aziz Zayed
McGill University
abd-el-aziz.zayed@mail.mcgill.ca

Jacob Mai Peng
McGill University
mai.peng@mail.mcgill.ca

Christophe Dubach
McGill University & MILA
christophe.dubach@mcgill.ca

Paper + GitHub link (Jonathan's web site)

EqSat Implementation Landscape

Custom Tweaks → Foresight

Specialized Saturation Processes

- Rule Scheduling
- Timeouts, Node Limits, etc.
- **Rebasing**
- **Multi-Phase Pipelines**
- **E-Graph Segmentation**

Saturation Strategies

+ Parallel E-Matching

Additional Metadata

- E-Class Analyses
- **Incremental EqSat**

Generalized Metadata

+ Slotted E-Graphs

Saturation Strategies: Vanilla

Custom EqSat Impl

```
def saturate_naive(egraph, rules):
  while True:
    egraph’ = rewrite(egraph, rules)
    if egraph’ == egraph:
      return egraph’

    egraph = egraph’
```

Traditional Library

```
saturate(
  egraph,
  rules,
  condition = UNTIL_SATURATION,
  scheduler =
    MAXIMAL_RULE_APPLICATION)
```

Foresight

```
strategy =
  MaximalRuleApplication(rules)
  .repeatUntilStable

strategy(egraph)
```

Saturation Strategies: Timeout + Backoff

Custom EqSat Impl

```
def saturate_backoff(
    egraph, rules,
    timeout, scheduler):
  i = 0
  end_time = now() + timeout
  while True:
    egraph’ = rewrite(
      egraph, scheduler(rules, i))

    if egraph’ == egraph
      or now() > end_time:

      return egraph’

    egraph = egraph’
    i += 1
```

Traditional Library

```
saturate(
  egraph,
  rules,
  condition = TIMEOUT(timeout),
  scheduler = BACKOFF(n, c))
```

Foresight

```
strategy =
  BackoffRuleApplication(
    rules, n, c)
  .withTimeout(timeout)
  .repeatUntilStable

strategy(egraph)
```

Saturation Strategies: Rebase + Timeout + Backoff

Custom EqSat Impl

```
def saturate_rebase(
    egraph, rules, iterations,
    condition, scheduler):
  for i in 0..iterations:
    j = 0
    while True:
      egraph’ = rewrite(
        egraph, scheduler(rules, j))

      if egraph’ == egraph
        or condition(egraph’):

        egraph = egraph’
        break
    j += 1
    expression = extract(egraph’)
    egraph = egraph_from(expression)
```

Traditional Library

```
def saturate_rebase(
    egraph, rules, iterations):
  for i in 0..iterations:
    egraph’ = saturate(
      egraph,
      rules,
      condition = TIMEOUT(timeout),
      scheduler = BACKOFF(n, c))

    expression = extract(egraph’)
    egraph = egraph_from(expression)

  return egraph
```

Foresight

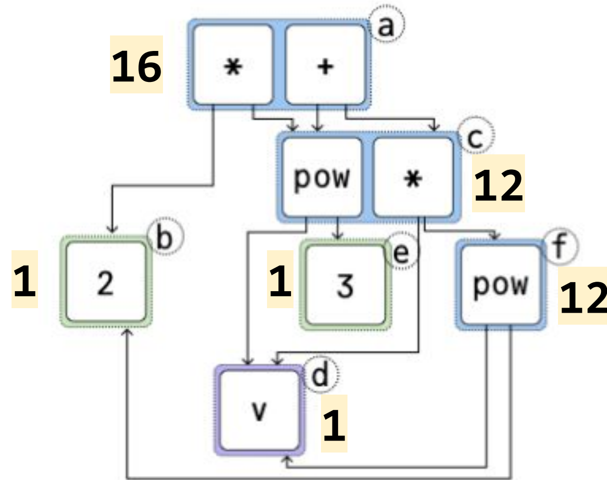
```
strategy =
  BackoffRuleApplication(
    rules, n, c)
  .withTimeout(timeout)
  .repeatUntilStable
  .thenRebase(extract)
  .withIterationLimit(iterations)
  .repeatUntilStable

strategy(egraph)
```

E-Class Analyses

- Derive fact from e-node → e.g., cost *c*
- Join facts → e.g., min(x, y)

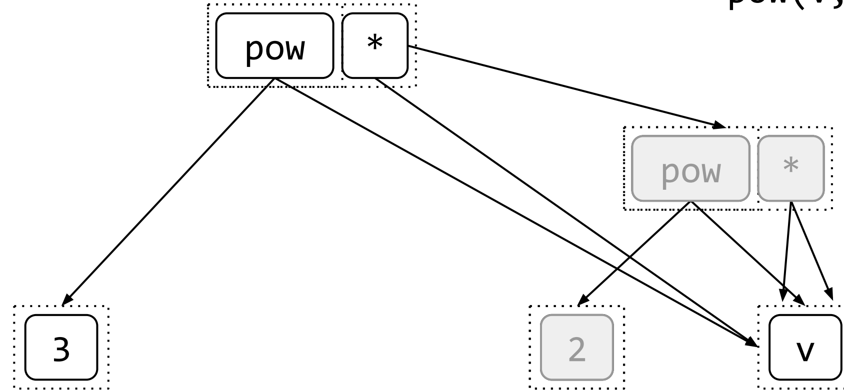
c(x + y) = 1 + *c*(x) + *c*(y)
c(x * y) = 3 + *c*(x) + *c*(y)
c(x pow y) = 10 + *c*(x) + *c*(y)
c(var or const) = 1

What if our metadata doesn't fit this scheme?



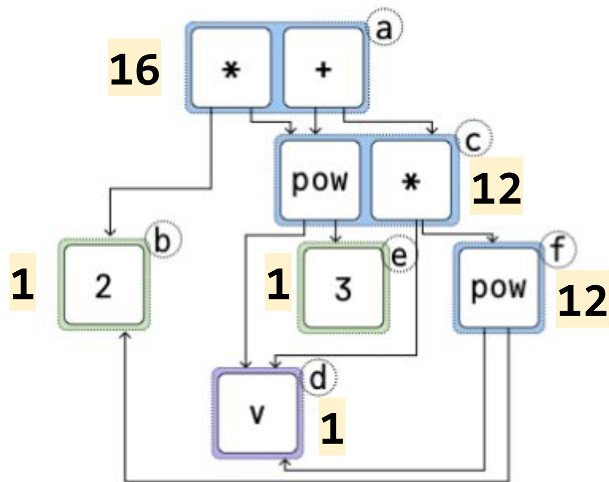
Incremental Equality Saturation

```
pow(v, 2) → v * v

pow(v, 3) → v * v * v
```

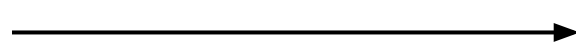


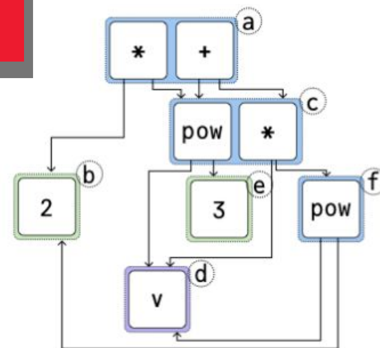
E-class from previous expression?
→ not an e-class analysis

Generalized Metadata



fact from e-node + join



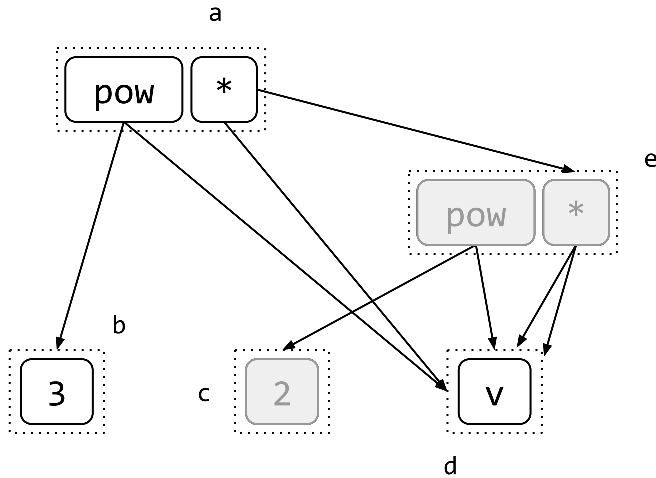


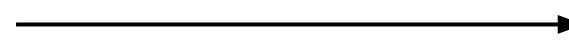
+

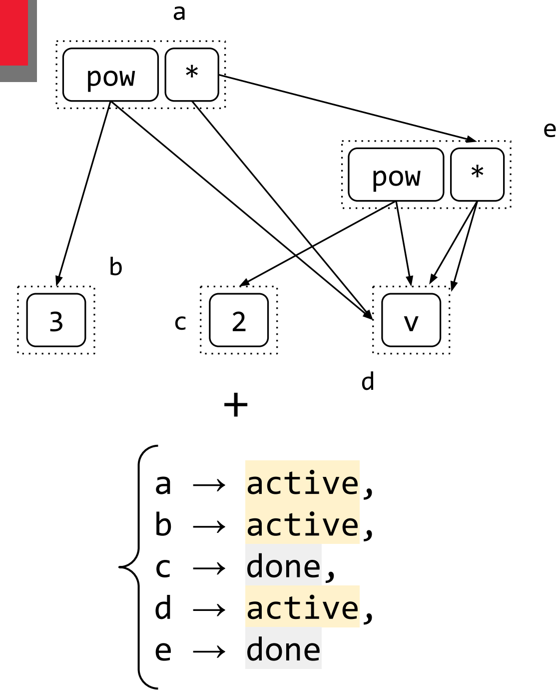
$$\left\{ \begin{array}{l} a \to 16, \\ b \to 1, \\ c \to 12, \\ d \to 1, \\ e \to 1, \\ f \to 12 \end{array} \right.$$

observe e-node adds +
observe e-class unions

Generalized Metadata







track active e-classes

observe e-node adds +
observe e-class unions

Generalized Metadata

```
strategy =
  MaximalRuleApplication(rules)
  .repeatUntilStable
  .addAnalyses(costAnalysis, extractionAnalysis, typeAnalysis)
```

Metadata Interface

```
interface Metadata:
  def onAddMany(added: Seq[(ENode, EClass)], after: EGraph) → Metadata
  def onUnionMany(equivalences: Set[Set[EClass]], after: EGraph) → Metadata
```

Figure 2

```
1 def rewrite_sequential(egraph, rules):
2   matches = []
3   for r in rules:
4     for ecls in egraph.classes:
5       case egraph.matchAndCompress(r, ecls) of
6         Some(Match(data)):
7           matches.append(Match(data))
8         None:
9           continue
10
11  for match in matches:
12    egraph.apply(match)
```

(a) Classic saturation process. **Highlighted** methods mutate e-graph data and must be executed sequentially.

```
1 def rewrite_parallel(egraph, rules):
2   matches = findMatches(rules, egraph.classes)
3   commands = computeUpdates(matches)
4   addCmds, unionCmds = simplifyAndBatch(
5     commands
6   )
7
8   return egraph.addMany(addCmds)
9                .unionMany(unionCmds)
```

(b) FORESIGHT's deferred saturation algorithm. **Highlighted** operations are fully parallel over their inputs. The `addMany` and `unionMany` methods are also partially parallelized.

Figure 2. Pseudo-code comparison of the rewriting algorithms. By default, equality saturation performs rewrites until fixpoint.

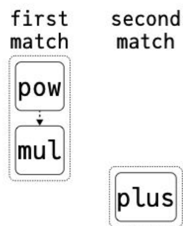
Figure 3

```
// first match
%0 = add (2)
%1 = add (pow d, %0)
%2 = add (mul d, %1)
union %2, c

// second match
%3 = add (plus c, c)
union %3, a
```

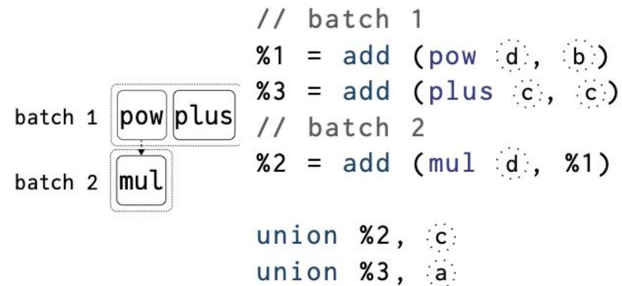
(a) Initial commands.

```
// first match
// (2) already in b
%1 = add (pow d, b)
%2 = add (mul d, %1)
union %2, c

// second match
%3 = add (plus c, c)
union %3, a
```

(b) After simplification.



(c) Dependencies for add commands.



(d) After batching. (e) Reordered commands.

Figure 3. Command reordering from the matches in Figure 1.

Figure 4

```
def saturate_naive(egraph, rules):
  while true:
    egraph' = rewrite(egraph, rules)
    if egraph' == egraph:
      return egraph'
```

(a) Classic saturation loop.

```
def saturate(egraph, rules, condition, scheduler):
  i = 0
  while true:
    egraph' = rewrite(egraph, scheduler(rules, i))
    if egraph' == egraph or condition(egraph):
      return egraph'
    i += 1
```

(b) Saturation with a scheduler and stopping condition.

```
strategy = MaximalRuleApplication(rules)
  .repeatUntilStable
strategy(egraph)
```

(c) Figure 4a as an FORESIGHT strategy.

```
strategy = BackoffRuleApplication(rules, n, c)
  .withTimeout(timeout)
  .withIterationLimit(iterationLimit)
  .repeatUntilStable
strategy(egraph)
```

(d) Figure 4b with backoff scheduler, as an FORESIGHT strategy.

Figure 4. Existing saturation behaviors and how they are expressed as strategies in FORESIGHT.

Figure 5

```
1 def withIterationLimit(inner_strat, limit):
2   i = 0
3   def apply(egraph):
4     i += 1
5     if i < limit:
6       return inner_strat(egraph)
7     return None
8   return apply
```

Figure 5. Pseudocode implementation of `withIterationLimit`.

Figure 6

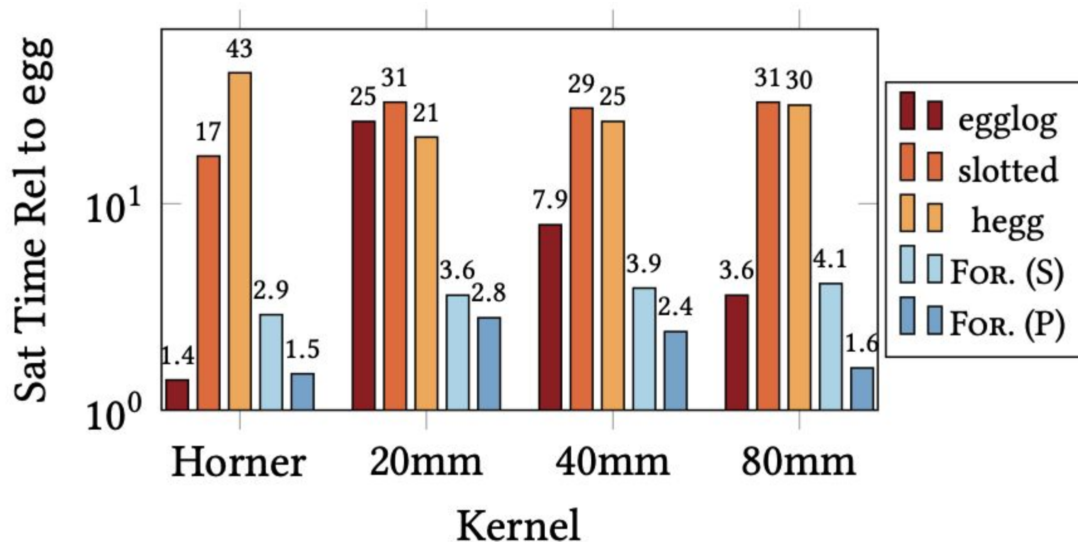


Figure 6. Comparison of saturation time across many implementations of equality saturation. For. (S) and For. (P) refer to sequential and parallel Foresight. The baseline is egg.

Figure 7

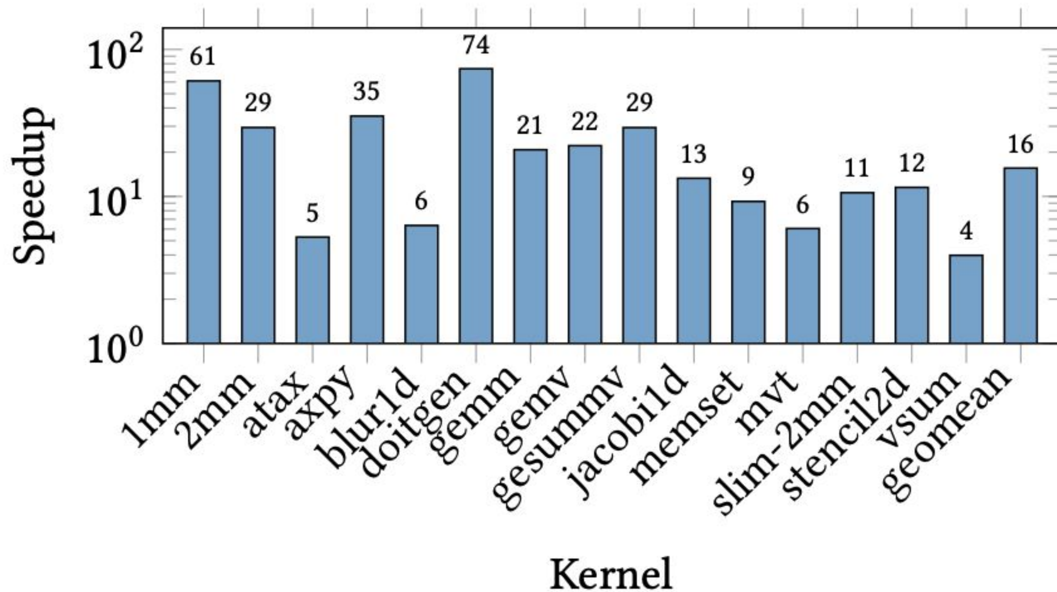


Figure 7. Saturation speedups of FORESIGHT relative to the original LIAR engine across BLAS benchmarks.

Figure 8

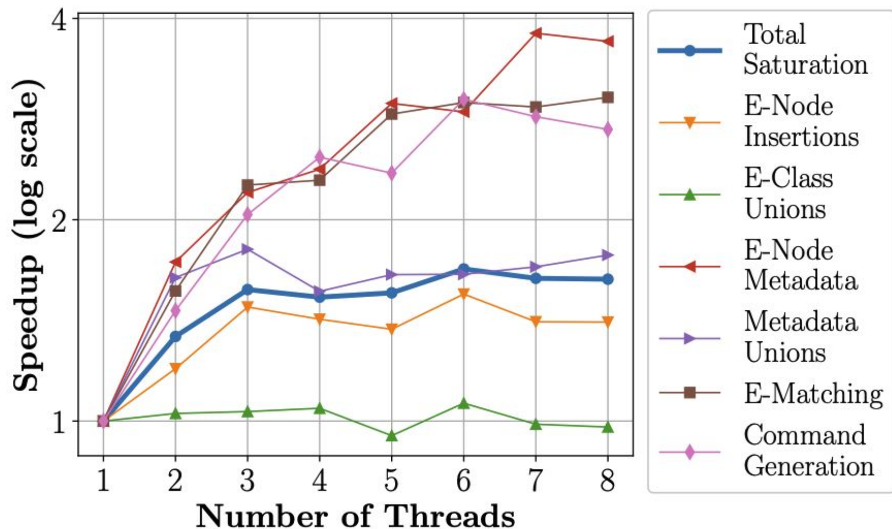


Figure 8. Parallel speedup across individual components of FORESIGHT's equality saturation engine for the stencil2d kernel. Rule matching and application scale effectively. Metadata updates scale partially, while hashconsing stays sequential.

Figure 9

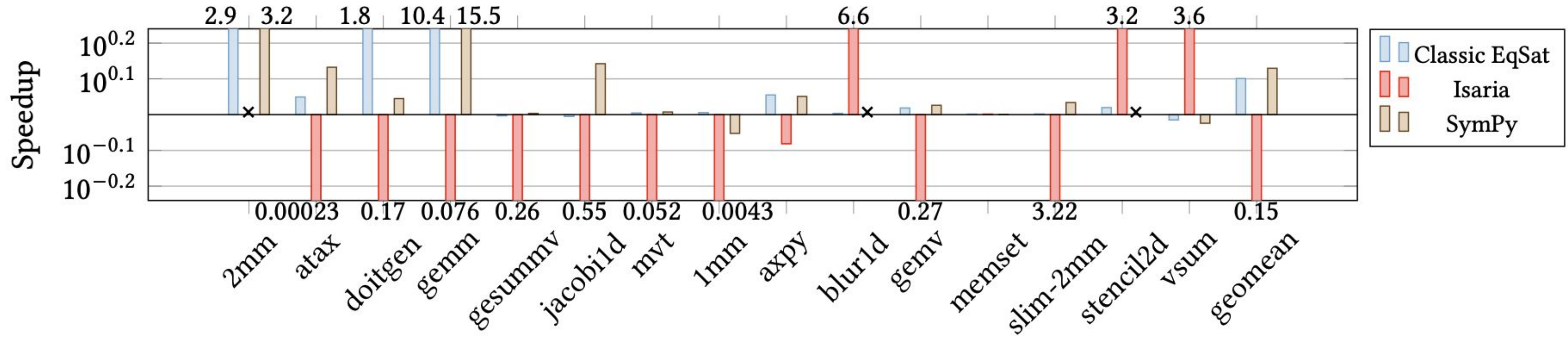


Figure 9. Run time speedup of FORESIGHT's solutions compared to baseline LIAR solutions. Each bar represents the ratio of the baseline run time and the FORESIGHT solution run time. Higher is better.

Figure 10

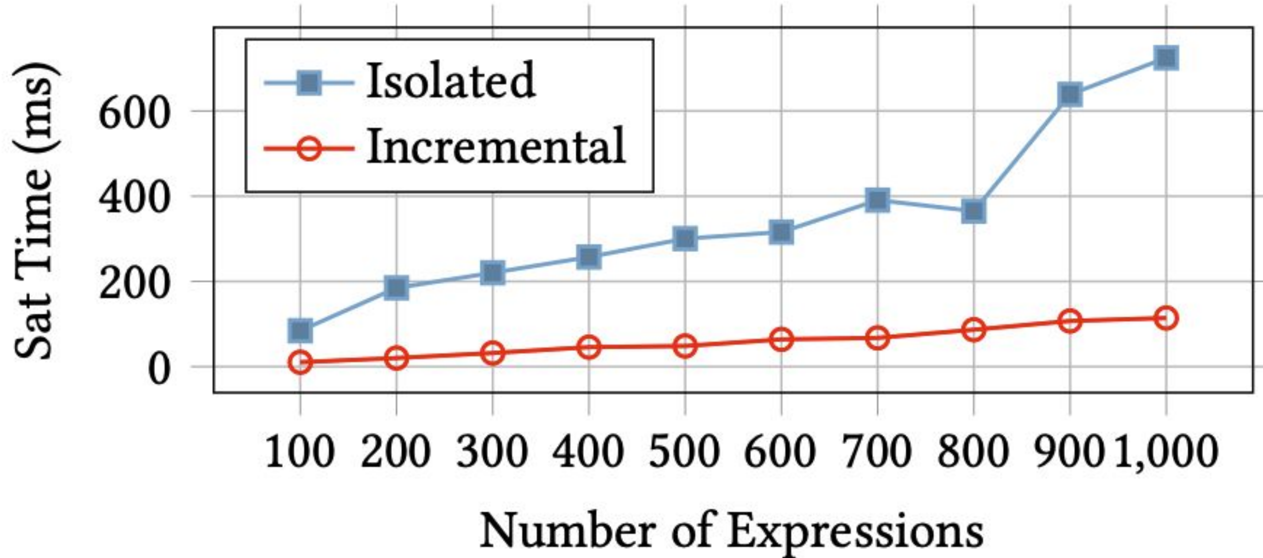


Figure 10. Saturation time of incremental vs standard EqSat. Inputs are identical; saturation uses one thread.

Table 1

Table 1. Memory usage comparison of the Java heap size used by Foresight and peak resident set size used by egg. Foresight is run in parallel with 8 threads.

Kernel	Foresight Heap Size	egg RSS
Horner	40 MB	4 MB
20mm	17 MB	3 MB
40mm	21 MB	4 MB
80mm	55 MB	13 MB

Table 2

Table 2. BLAS idioms where FORESIGHT finds a better solution than LIAR. In all other cases, the solutions are identical. Benchmark run time speedups (*Sp.*) vary from 1× to 10×.

Kernel	LIAR solution	FORESIGHT solution	Sp.
2mm	3 × axpy + 2 × gemv + 4 × memset	2 × gemm + 1 × memset + 1 × transpose	3×
gemm	3 × axpy + 1 × gemv + 3 × memset	1 × gemm + 1 × transpose	10×
slim-2mm	1 × gemm + 1 × gemv + 2 × memset + 1 × transpose	2 × gemm + 2 × memset + 2 × transpose	1×