Compiler Design

An Introduction to Equality Saturation and Its Applications

March 25, 2026

Abd-El-Aziz Zayed

McGill

About Me

- Undergrad in SWE (2020–2024)
- Summer research in CASL Lab (2024)
- MSc in CASL Lab (2025)
- Fast-track to PhD in CASL Lab (2026–)
- Equality saturation in production compilers
- azizzayed.com

Outline

Compiler Optimization

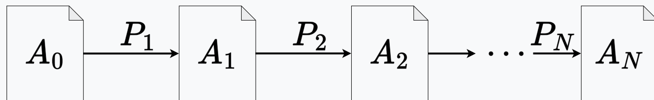
Equality Saturation

Case Studies

Applications

Pass Sequences

An optimizer performs a sequence of passes (a.k.a. rewrites, transformations) on a program to improve its performance.



Optimization Passes

- Constant folding
- Dead code elimination
- Common subexpression elimination

- Loop unrolling
- Loop fusion
- Loop tiling
- Loop vectorization

Optimization Passes

- Constant folding
- Dead code elimination
- Common subexpression elimination

Before

```
int arr[2 * 50];
int x = 3* (2 + 7);
```

After Constant Folding

```
int arr[100];
int x = 27;
```

Optimization Passes

- Constant folding
- Dead code elimination
- Common subexpression elimination

Before

```
int debug = 0;
if (debug)
  log("debug"); // never runs
x = compute();
```

After Dead Code Elimination

```
x = compute();
```

Optimization Passes

- Constant folding
- Dead code elimination
- Common subexpression elimination

Before

```
a = x * y + z;
b = x * y - z;
```

After CSE

```
t = x * y; // computed once
a = t + z;
b = t - z;
```

Optimization Passes

Before

```
for (i = 0; i < 4; i++)
  a[i] = b[i] + c[i];
```

After Loop Unrolling

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
```

- Loop unrolling
- Loop fusion
- Loop tiling
- Loop vectorization

Optimization Passes

Before

```
for (i = 0; i < n; i++)
  a[i] = b[i];
for (i = 0; i < n; i++)
  c[i] = a[i];
```

After Loop Fusion

```
for (i = 0; i < n; i++) {
  a[i] = b[i];
  c[i] = a[i];
}
```

- Loop unrolling
- Loop fusion
- Loop tiling
- Loop vectorization

Optimization Passes

Before

```
int N = 1024; // matrix size
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    C[i][j] += A[i][j];
```

After Loop Tiling

```
int N = 1024; // matrix size
int T = 32; // tile size
for (ii=0; ii<N; ii+=T)
  for (jj=0; jj<N; jj+=T)
    for (i=ii; i<ii+T; i++)
      for (j=jj; j<jj+T; j++)
        C[i][j] += A[i][j];
```

- Loop unrolling
- Loop fusion
- Loop tiling
- Loop vectorization

Optimization Passes

Before

```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

After Loop Vectorization

```
for (i = 0; i < n; i += 4) {
  va = _mm_load_ps(&a[i]);
  vb = _mm_load_ps(&b[i]);
  _mm_store_ps(&c[i],
       _mm_add_ps(va, vb));
}
```

- Loop unrolling
- Loop fusion
- Loop tiling
- Loop vectorization

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \to x(y/z)}$
- P_2. $\underline{x \cdot 2 \to x \ll 1}$
- P_3. $\underline{xy \to yx}$
- P_4. $\underline{x/x \to 1}$
- P_5. $\underline{x \cdot 1 \to x}$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \to x(y/z)}$
- P_2. $\underline{x \cdot 2 \to x \ll 1}$
- P_3. $\underline{xy \to yx}$
- P_4. $\underline{x/x \to 1}$
- P_5. $\underline{x \cdot 1 \to x}$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $xy/z \rightarrow x(y/z)$
- P_2. $x \cdot 2 \rightarrow x \ll 1$
- P_3. $xy \rightarrow yx$
- P_4. $x/x \rightarrow 1$
- P_5. $x \cdot 1 \rightarrow x$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \to x(y/z)}$
- P_2. $\underline{x \cdot 2 \to x \ll 1}$
- P_3. $\underline{xy \to yx}$
- P_4. $\underline{x/x \to 1}$
- P_5. $\underline{x \cdot 1 \to x}$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \rightarrow x(y/z)}$
- P_2. $\underline{x \cdot 2 \rightarrow x \ll 1}$
- P_3. $\underline{xy \rightarrow yx}$
- P_4. $\underline{x/x \rightarrow 1}$
- P_5. $\underline{x \cdot 1 \rightarrow x}$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2 \xrightarrow{P_1} a \cdot (2/2)$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \rightarrow x(y/z)}$
- P_2. $\underline{x \cdot 2 \rightarrow x \ll 1}$
- P_3. $\underline{xy \rightarrow yx}$
- P_4. $\underline{x/x \rightarrow 1}$
- P_5. $\underline{x \cdot 1 \rightarrow x}$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2 \xrightarrow{P_1} a \cdot (2/2) \xrightarrow{P_4} a \cdot 1$

Phase Ordering Problem

The order in which passes are applied matters. Applying passes in the wrong order may block further applicable rules, missing the global optimum [1].

Rewrite rules:

- P_1. $\underline{xy/z \rightarrow x(y/z)}$
- P_2. $\underline{x \cdot 2 \rightarrow x \ll 1}$
- P_3. $\underline{xy \rightarrow yx}$
- P_4. $\underline{x/x \rightarrow 1}$
- P_5. $\underline{x \cdot 1 \rightarrow x}$

Applied to the expression $(a \cdot 2)/2$:

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2 \xrightarrow{P_1} a \cdot (2/2) \xrightarrow{P_4} a \cdot 1 \xrightarrow{P_5} a$

Phase Ordering Problem

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2 \xrightarrow{P_1} a \cdot (2/2) \xrightarrow{P_4} a \cdot 1 \xrightarrow{P_5} a$

1. The optimizer may get stuck in a local optimum if it applies transformations out-of-order (e.g. P_2 before P_1).
2. The passes are destructive: we lose the original and intermediate versions of the expression.

Phase Ordering Problem

- $(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2$
- $(a \cdot 2)/2 \xrightarrow{P_1} a \cdot (2/2) \xrightarrow{P_4} a \cdot 1 \xrightarrow{P_5} a$

1. The optimizer may get stuck in a local optimum if it applies transformations out-of-order (e.g. P_2 before P_1).
2. The passes are destructive: we lose the original and intermediate versions of the expression.

- $$(a \cdot 2)/2 \xrightarrow{P_2} (a \ll 1)/2 \xrightarrow{P_1} a \cdot (2/2) \xrightarrow{P_4} a \cdot 1 \xrightarrow{P_5} a$$

Outline

Compiler Optimization

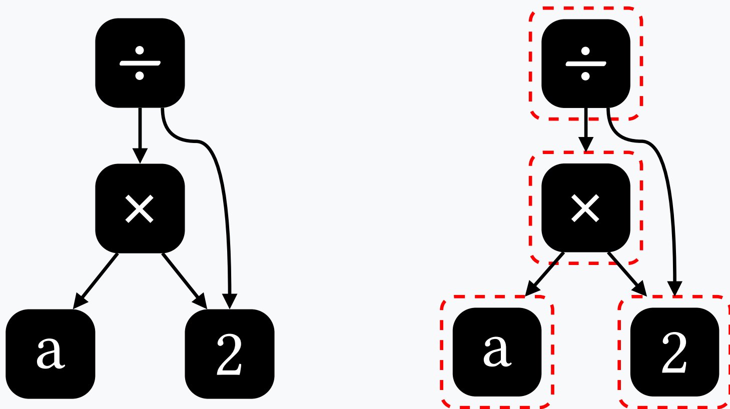
Equality Saturation

Case Studies

Applications

E-Graphs

An **e**-graph stores many equivalent forms of an expression [1].

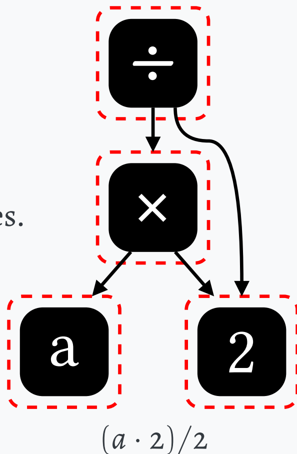


$(a \cdot 2)/2$

E-Graphs

An **e**-graph stores many equivalent forms of an expression [1].

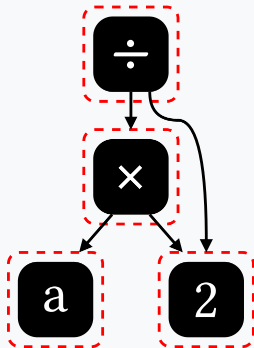
1. e-node : Operation node.
2. **e-class** : Set of equivalent e-nodes.
3. **e-graph**: Set of e-classes.



$(a \cdot 2)/2$

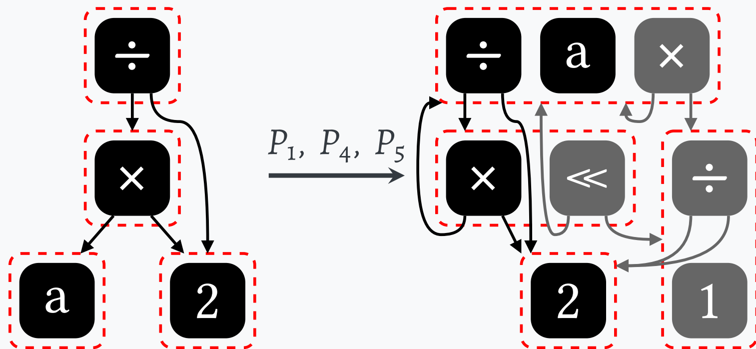
Apply Rewrite Rules to an E-Graph

- P_1. $xy/z \to x(y/z)$
- P_2. $x \cdot 2 \to x \ll 1$
- P_3. $xy \to yx$
- P_4. $x/x \to 1$
- P_5. $x \cdot 1 \to x$

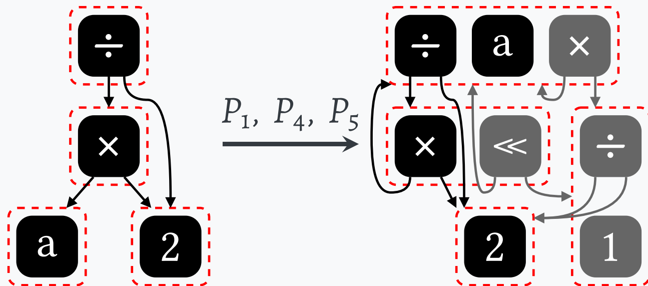


Apply Rewrite Rules to an E-Graph

- P_1. $xy/z \to x(y/z)$
- P_2. $x \cdot 2 \to x \ll 1$
- P_3. $xy \to yx$
- P_4. $x/x \to 1$
- P_5. $x \cdot 1 \to x$



Equality Saturation



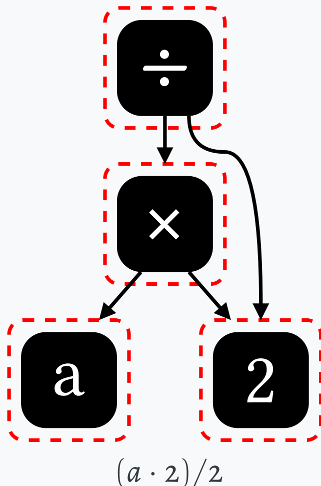
Problems are solved:

1. The global optimum is found by extracting the optimal expression from the e-graph via a cost model.
2. The original and intermediate versions of the program are preserved in the e-graph.

Equality Saturation

Equality saturation steps [1]:

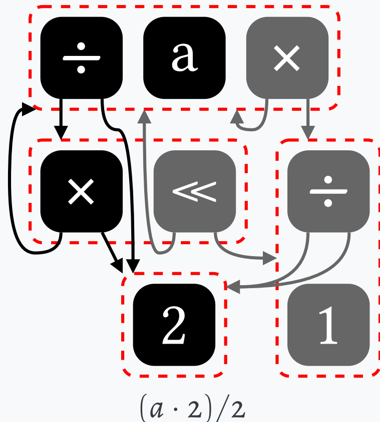
1. Build an **e**-graph of $(a \cdot 2)/2$.
2. Apply the rewrite rules to the e-graph until a fixed-point.
3. Extract the optimized expression via a cost model.



Equality Saturation

Equality saturation steps [1]:

1. Build an **e**-graph of $(a \cdot 2)/2$.
2. Apply the rewrite rules to the e-graph until a fixed-point.
3. Extract the optimized expression via a cost model.



Tree Cost Extraction

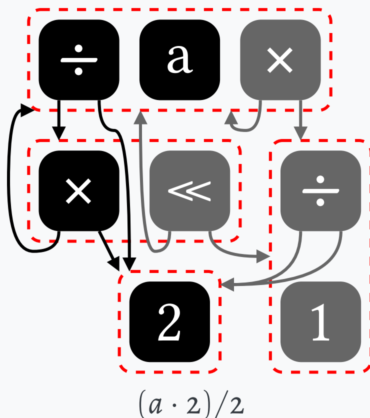
We define a **cost model** C to extract the optimal expression from the saturated e-graph.

$$C(n) = \begin{cases} 10 & \text{if } n \text{ is } \times \\ 20 & \text{if } n \text{ is } \div \\ 5 & \text{if } n \text{ is } \ll \\ 1 & \text{else} \end{cases}$$

$$C(E) = \min_{n \in E} C(T_n)$$

$$C(T_n) = C(n) + \sum_{n \to E} C(E)$$

Where n is an e-node, E is an e-class, T_n is the tree rooted at e-node n, and $\to$ is an e-node to e-class edge.



Tree Cost Extraction

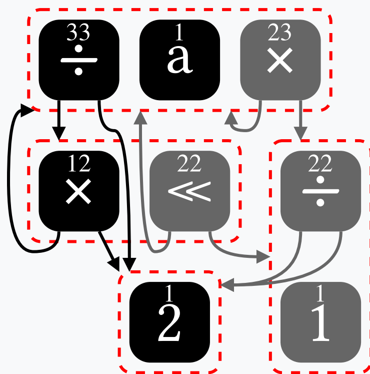
We define a **cost model** C to extract the optimal expression from the saturated e-graph.

$$C(n) = \begin{cases} 10 & \text{if } n \text{ is } \times \\ 20 & \text{if } n \text{ is } \div \\ 5 & \text{if } n \text{ is } \ll \\ 1 & \text{else} \end{cases}$$

$$C(E) = \min_{n \in E} C(T_n)$$

$$C(T_n) = C(n) + \sum_{n \to E} C(E)$$

Where n is an e-node, E is an e-class, T_n is the tree rooted at e-node n, and $\to$ is an e-node to e-class edge.



Outline

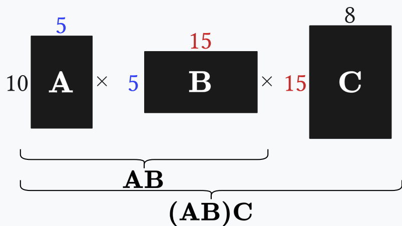
Compiler Optimization

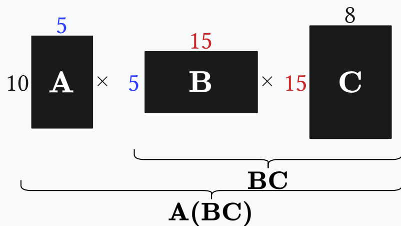
Equality Saturation

Case Studies

Applications

MatMul Associativity [2]

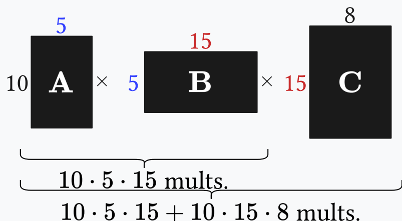


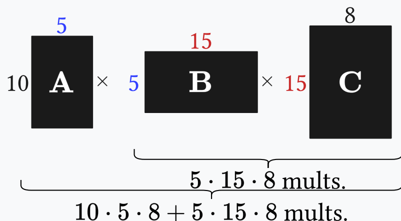


$$\mathbf{(AB)\,C} = \mathbf{A\,(BC)}$$

How do we find out which one is more efficient?

The Order of Operations Matters



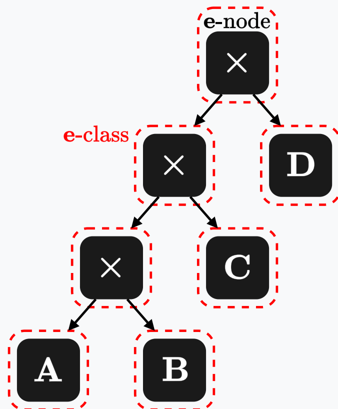


`cost` $\mathbf{(AB)\,C} = 1950$ multiplications.

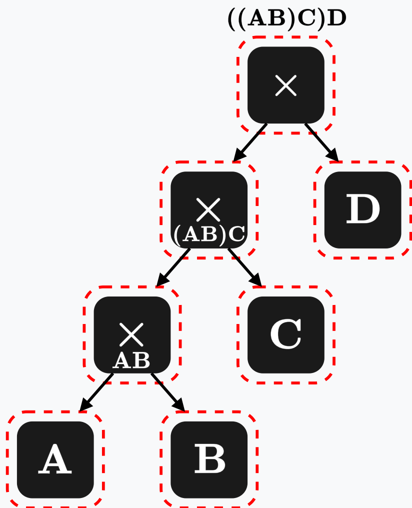
`cost` $\mathbf{A\,(BC)} = 1000$ multiplications.

Equality Saturation for ((AB) C) D

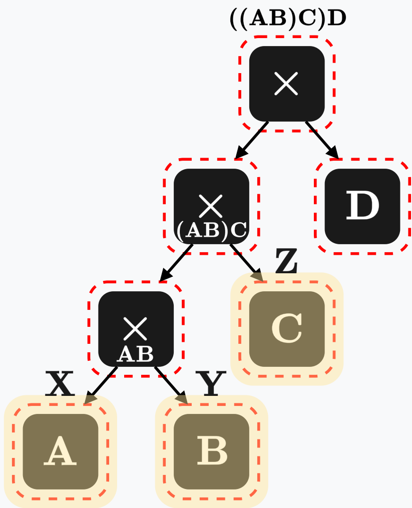
1. Build an **e**-graph of **((AB) C) D**.
2. Apply the rewrite rule **(XY) Z** ⇔ **X (YZ)** to the e-graph until a fixed-point.
3. Extract the optimized expression via a cost model.



2. Apply the Rewrite (XY) Z ⇔ X (YZ)

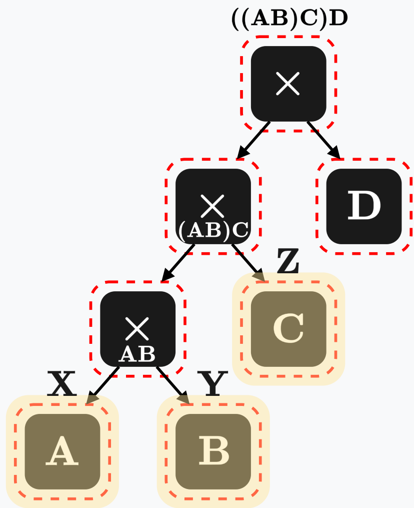


2. Apply the Rewrite (XY) Z ⇔ X (YZ)

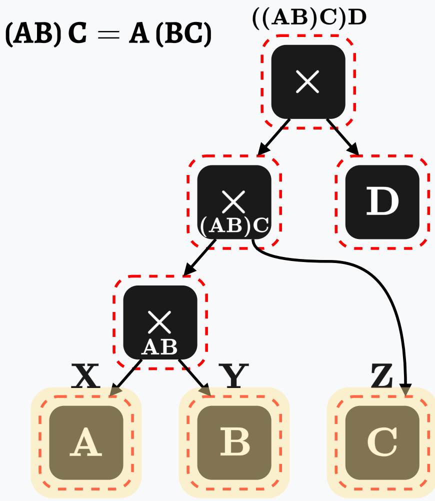


$$(AB)\,C = A\,(BC)$$

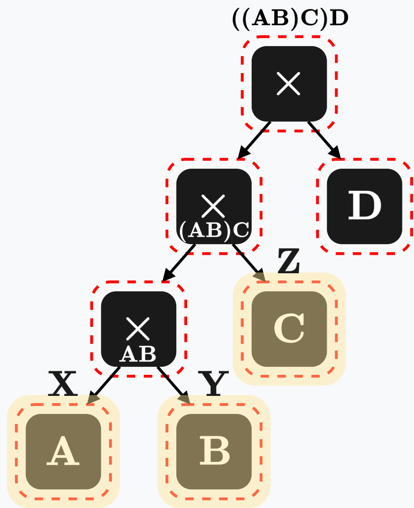
2. Apply the Rewrite (XY) Z $\Leftrightarrow$ X (YZ)



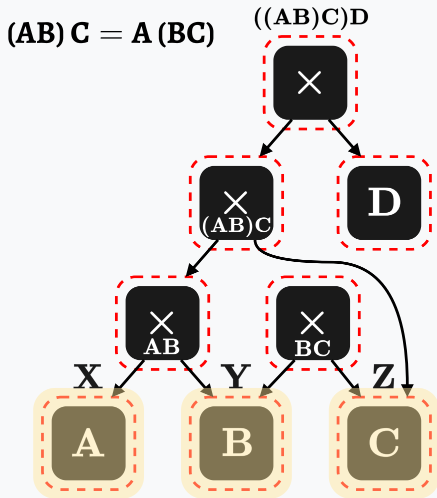
(AB) C = A (BC)



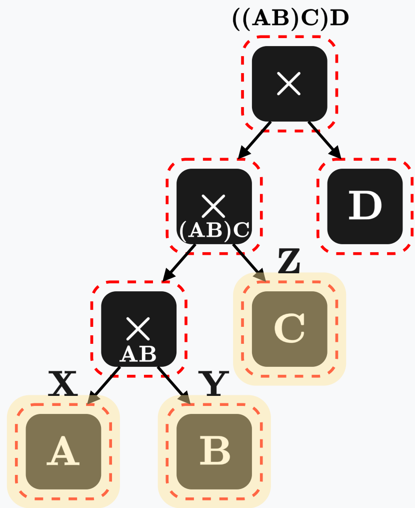
2. Apply the Rewrite (XY) Z ⇔ X (YZ)



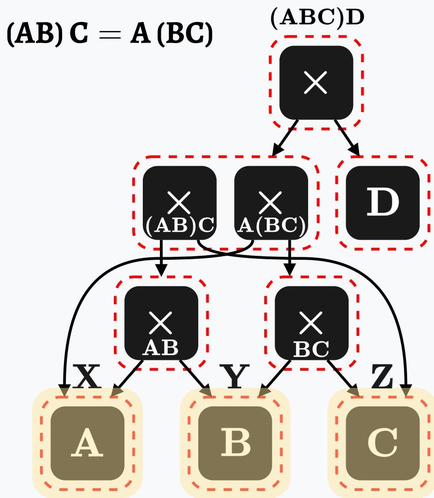
(AB) C = A (BC)



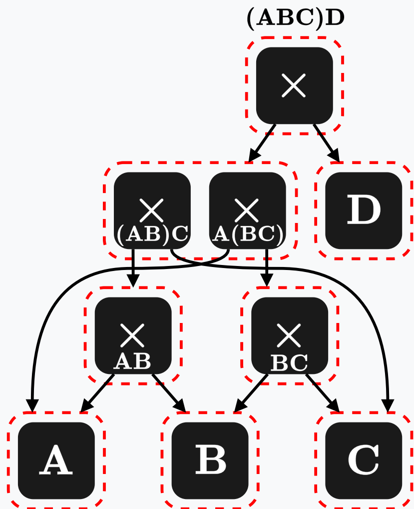
2. Apply the Rewrite (XY) Z ⇔ X (YZ)



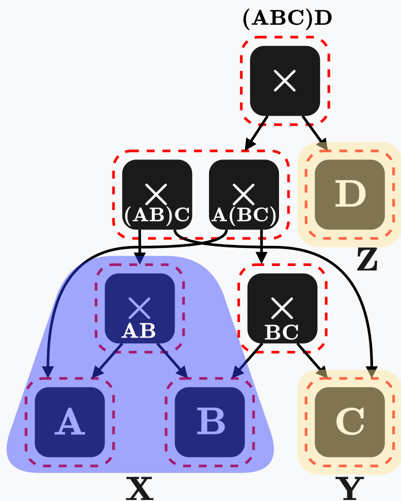
(AB) C = A (BC)



2. Apply the Rewrite (XY) Z ⇔ X (YZ)



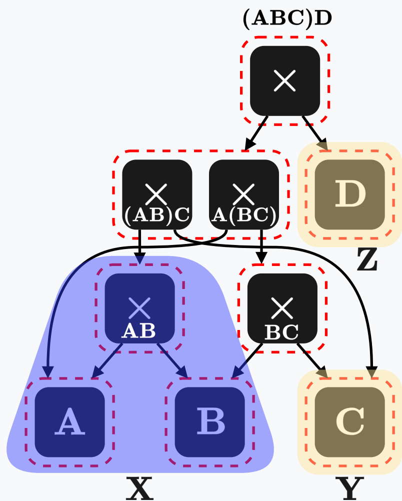
2. Apply the Rewrite (XY) Z ⇔ X (YZ)

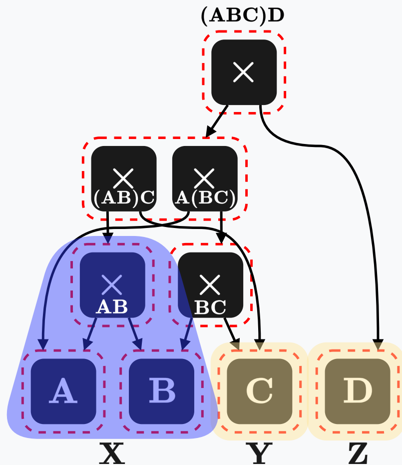


$$((AB)\,C)\;D = (AB)\,(CD)$$

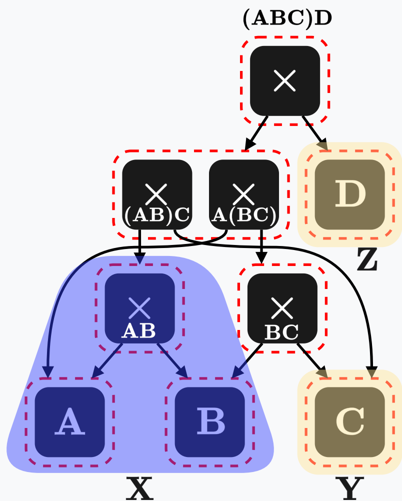
2. Apply the Rewrite (XY) Z $\Leftrightarrow$ X (YZ)

((AB) C) D = (AB) (CD)

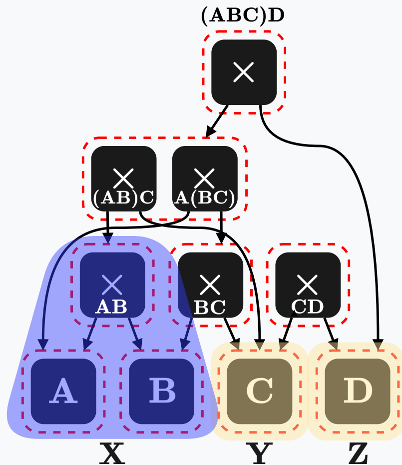




2. Apply the Rewrite (XY) Z $\Leftrightarrow$ X (YZ)

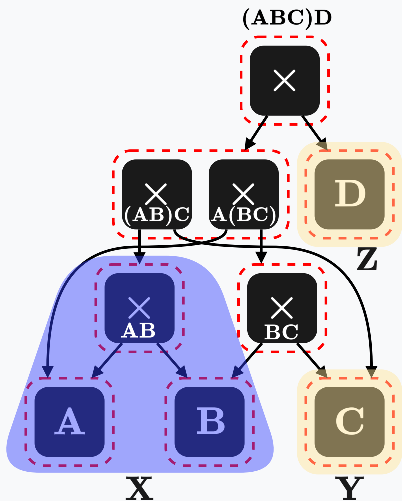


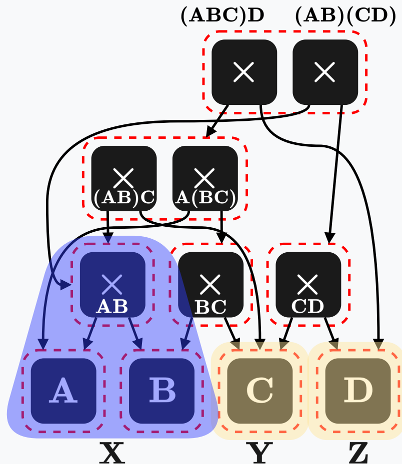
((AB) C) D = (AB) (CD)



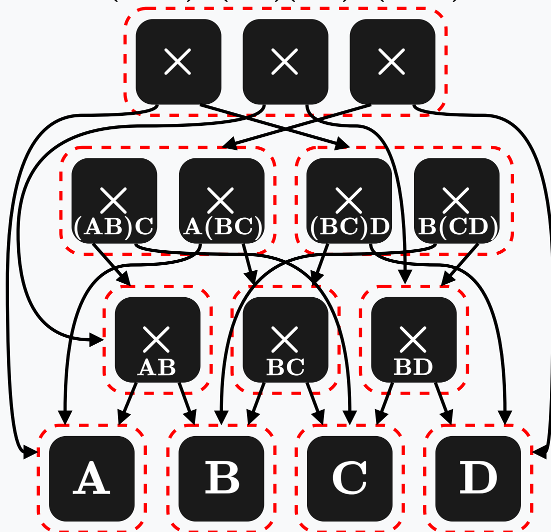
2. Apply the Rewrite (XY) Z ⇔ X (YZ)

$((AB)\,C)\;D = (AB)\,(CD)$

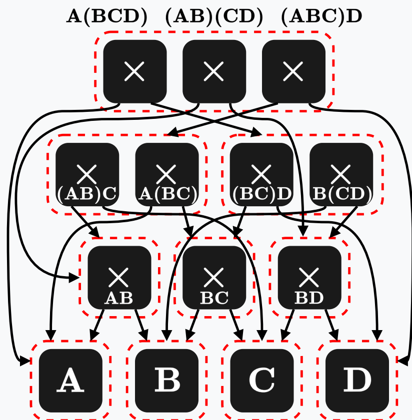




Final E-Graph After Saturation



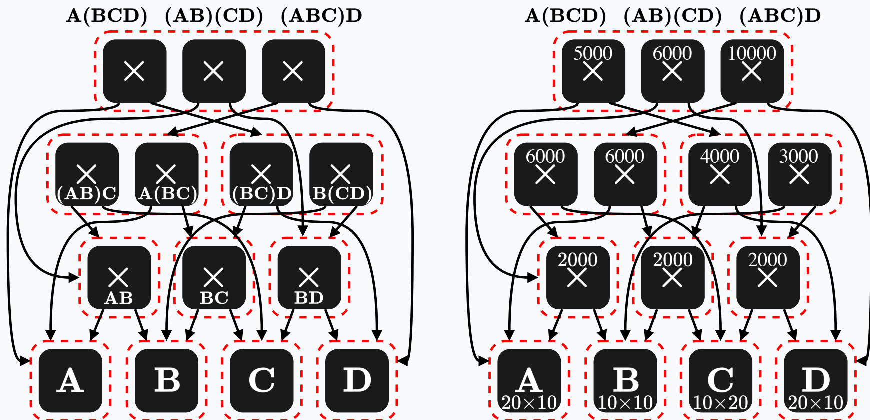
3. Extract the Optimal Expression



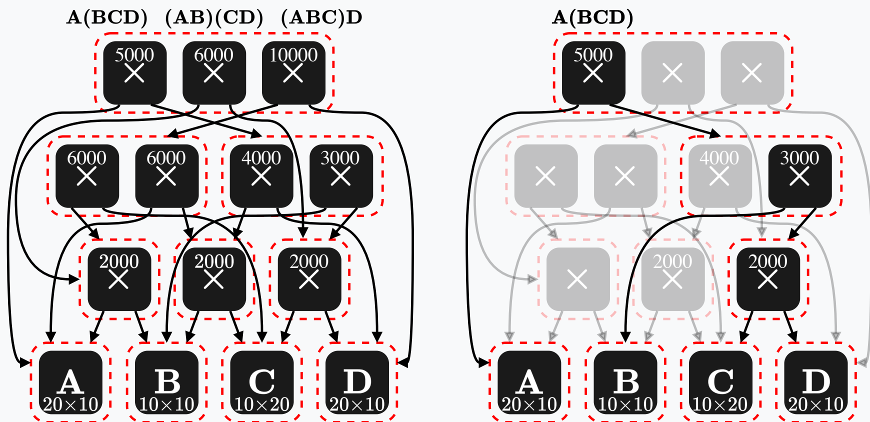
- Root e-class contains equivalent expressions of the program.
- Compute cost of e-nodes in the root e-class.
- Pick the e-node in the root e-class with the lowest cost.

3. Extract the Optimal Expression

$$\texttt{cost}\ \mathbf{XY} = \text{nrows}(\mathbf{X}) \cdot \text{ncols}(\mathbf{X}) \cdot \text{ncols}(\mathbf{Y})$$



The Optimal Expression is **A (B (CD))**



Polynomial Evaluation: Horner's Method

Reduce from $O(n^2)$ to $O(n)$ multiplications [2].

$$
\begin{aligned}
P(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\
&= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)))
\end{aligned}
$$

What rewrites do we need to perform Horner's method?

Polynomial Evaluation: Horner's Method

Reduce from $O(n^2)$ to $O(n)$ multiplications [2].

$$\begin{aligned} P(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n))) \end{aligned}$$

What rewrites do we need to perform Horner's method?

- Exponentiation: $\underline{x^0 \Rightarrow 1}$ and $\underline{x^n \Rightarrow x \cdot x^{n-1}}$

Polynomial Evaluation: Horner's Method

Reduce from $O(n^2)$ to $O(n)$ multiplications [2].

$$
\begin{aligned}
P(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\
&= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)))
\end{aligned}
$$

What rewrites do we need to perform Horner's method?

- Exponentiation: $x^0 \Rightarrow 1$ and $x^n \Rightarrow x \cdot x^{n-1}$
- Commutativity: $x + y \Leftrightarrow y + x$ and $xy \Leftrightarrow yx$
- Associativity: $(x + y) + z \Leftrightarrow x + (y + z)$ and $(xy)z \Leftrightarrow x(yz)$
- Distributivity: $x(y + z) \Leftrightarrow xy + x \cdot z$
- Identity: $x \cdot 1 \Rightarrow x$

Outline

Compiler Optimization

Equality Saturation

Case Studies

Applications

The egg Library

- **egg** is an efficient EqSat implementation in Rust [1].
- Fast: union-find with path compression & union by rank ($\approx O(1)$ amortized)
- Flexible: pluggable cost functions and analysis
- Used in majority of EqSat research papers

Applications of Equality Saturation & E-Graphs

Compiler Optimization: Herbie (floating-point accuracy), RISE (loop optimizations), RTL synthesis

Applications of Equality Saturation & E-Graphs

Compiler Optimization: Herbie (floating-point accuracy), RISE (loop optimizations), RTL synthesis

Program Synthesis: Ruler (rewrite rule generation), Hardware mapping, Decompilation

Applications of Equality Saturation & E-Graphs

Compiler Optimization: Herbie (floating-point accuracy), RISE (loop optimizations), RTL synthesis

Program Synthesis: Ruler (rewrite rule generation), Hardware mapping, Decompilation

Automated Reasoning: SMT solvers (Z3, cvc5), Theorem proving (Lean 4), Bug detection

Applications of Equality Saturation & E-Graphs

Compiler Optimization: Herbie (floating-point accuracy), RISE (loop optimizations), RTL synthesis

Program Synthesis: Ruler (rewrite rule generation), Hardware mapping, Decompilation

Automated Reasoning: SMT solvers (Z3, cvc5), Theorem proving (Lean 4), Bug detection

Numerical Computing: TenSat (tensor graph optimization), Deep learning inference, Linear algebra

Applications of Equality Saturation & E-Graphs

Compiler Optimization: Herbie (floating-point accuracy), RISE (loop optimizations), RTL synthesis

Program Synthesis: Ruler (rewrite rule generation), Hardware mapping, Decompilation

Automated Reasoning: SMT solvers (Z3, cvc5), Theorem proving (Lean 4), Bug detection

Numerical Computing: TenSat (tensor graph optimization), Deep learning inference, Linear algebra

Other Domains: Query optimization, Julia IR optimization

Key insight: Wherever we have rewrite systems and need to find optimal solutions, e-graphs help explore the search space effectively.

Limitations of Equality Saturation

Despite its power, equality saturation has important limitations:

- **Search space explosion:** E-graphs can grow exponentially with complex rewrite rules and large programs.
- **Scalability challenges:** Memory usage and time to saturation can be prohibitive for large programs. Moreover, extraction is NP-hard in general.
- **Cost model design:** Finding the right cost model for extraction is domain-specific and non-trivial.
- **Termination guarantees:** Not all rewrite systems will terminate or saturate in reasonable time.

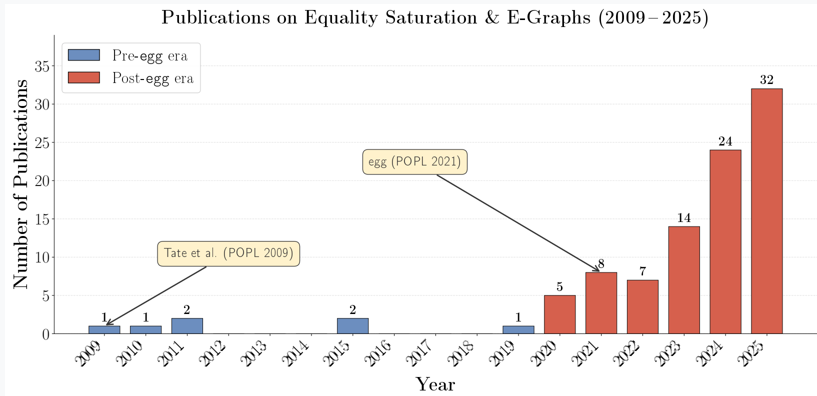
Active research: Guided saturation, pruning strategies, and heuristics aim to address these challenges.

Scalability Study [2]

Bench.	#Rules	#Ops	Saturation	Canonicalization
Img Conv	1	29	<0.1ms	0.2ms
Vec Norm	1	44	<0.1ms	0.2ms
Poly	8	26	2ms	0.2ms
3MM	5	8	1ms	0.1ms
10MM	5	22	4ms	0.1ms
20MM	5	42	23ms	0.2ms
40MM	5	82	235ms	0.3ms
80MM	5	162	3732ms	0.6ms

Research Trends: Growing Field

EqSat research growth since the original 2009 paper [1].



Equality Saturation in the CASL Lab

PI Christophe Dubach

Jonathan Van der Cruysse
Latent idiom recognition using EqSat & Foresight

Tzung-Han Juang
EqSat for hardware resource sharing

Adam Musa
Driving EqSat with RL

Aziz Zayed (me)
EqSat and egraphs in modern compilers like MLIR

Research Opportunities as an Undergrad

- COMP 400 research project
- Summer internship in the CASL lab
- NSERC USRA (~$8000) — check if deadline has passed

References

[1] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09, Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 264–276. DOI: 10.1145/1480881.1480915. [Online]. Available: https://doi.org/10.1145/1480881.1480915.

[2] A.-E.-A. Zayed and C. Dubach, "Dialegg: Dialect-agnostic mlir optimizer using equality saturation with egglog," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25, Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 271–283. DOI: 10.1145/3696443.3708957. [Online]. Available: https://doi.org/10.1145/3696443.3708957.