E-Graphs and their applications

Mihir Tandon

4/20/26

Outline

Disclaimer

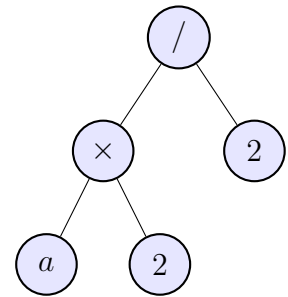
- This presentation is not completely formal, but instead meant to be a gentle introduction to E-graphs and how they are used.

Section 1

Introduction

Abstract Syntax and ASTs

- In compilers and formal systems, programs are often represented as trees.
- Abstract Syntax Trees (ASTs) give us a clear, recursive structure:
  - **Leaves:** Constants, variables (e.g., 2, a).
  - **Internal Nodes:** Operators (e.g., $+$, $\times$, $/$) acting on children.
- Example: The expression can be expressed as $(a \times 2)/2$ is an AST with $/$ at the root.
- While simple to traverse, standard trees are entirely rigid. One tree represents exactly one specific syntactic structure.



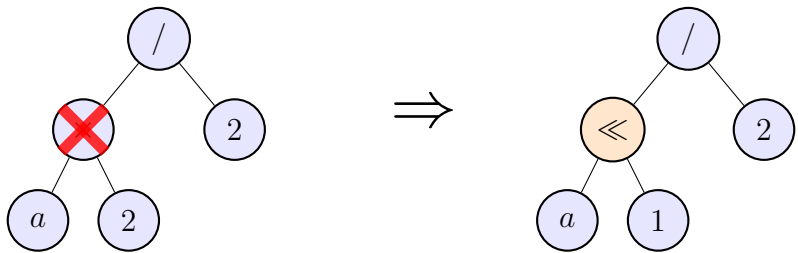
Traditional Term Rewriting

- Compilers heavily rely on rewriting these ASTs to optimize code.
- **Example Rule 1:** $x \times 2 \to x \ll 1$ (Bit shift is cheaper).
- **Example Rule 2:** $(x \times y)/y \to x$ (Algebraic simplification).
- **Problem:** Traditional rewriting is fundamentally *destructive*.
- When we apply a rule, we overwrite the old tree with the new one.
- If we start with $(a \times 2)/2$, applying Rule 1 gives $(a \ll 1)/2$.
- But now Rule 2 can no longer match! We are stuck with a suboptimal expression.

Visualizing Destructive Rewriting

- Apply Rule($x \times 2 \to x \ll 1$): The original $\times$ subtree is physically destroyed and overwritten by the new $\ll$ structure.

- We are now stuck. Rule 2 ($(x \times y)/y \to x$) can no longer "see" the $\times$ node it requires to match.



The Phase Ordering Problem

- This destructive nature leads directly to the **Phase Ordering Problem**.
- If we apply Optimization A before Optimization B, we might permanently lose the opportunity to apply B.
- If we apply B before A, we might lose A.
- In a real compiler with hundreds of heuristics, finding the "optimal" sequence of rules is essentially an intractable search problem.
- We need a data structure that remembers history and explores all paths simultaneously.

The E-Graph

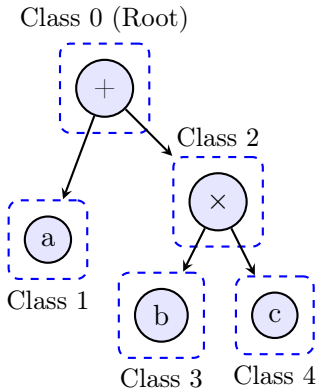
- An **E-Graph** (Equivalence Graph) solves this by compactly representing exponentially many equivalent expressions at once.
- It consists of two core concepts:
  - **E-Nodes:** Represent operations (just like AST nodes).
  - **E-Classes:** Sets of E-nodes that have been proven to be equivalent.
- Crucially, children of E-nodes point to *E-classes*, not other E-nodes!
- Invented for use in automated theorem provers by Greg Nelson and Derek C. Oppen, but didn't gain widespread use in compilers until 2021.

E-Graph Example

Initial Expression:

$$a + (b \times c)$$

- Before any rewrites, this looks like a standard tree.
- We map this structure 1-to-1 into an E-Graph.
- Every sub-expression gets its own distinct E-Class (Indicated by dashes).



Equality Saturation

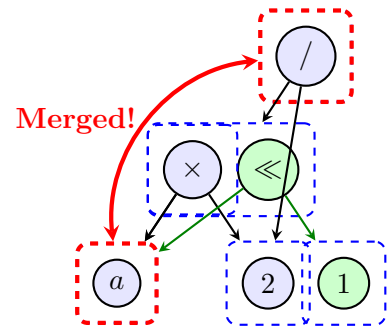
- When applying a rewrite rule, instead of replacing a matched pattern A with B, we simply *add* B into the exact same E-class as A. Both variants are preserved simultaneously.

- The engine repeatedly searches for patterns and applies rewrite rules across the entire E-graph in a process known as **equality saturation**. This loop continues until no new expressions can be discovered (saturation) or a resource limit is reached.

E-graphs/Equality Saturation Example

- **Step 1 (Match):** Start with initial E-graph (AST) for $(a \times 2)/2$. Each node is in its own E-class (dashed boxes).
- **Step 2 (Apply):** Apply rewrite rule $x \times 2 \to x \ll 1$ and insert $a \ll 1$. The E-class containing $\times$ now *also* contains $\ll$.
- **Step 3 (Merge):** We use the rule $(x \times y)/y \to x$. We *merge* the root E-class with the E-class containing a.

Result: The E-graph remembers *both* paths. We avoided phase ordering and found the global optimum (a)!

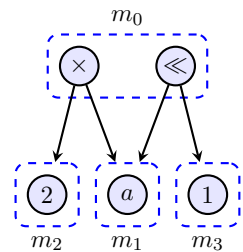


Formalizing the E-Graph

- We define an E-graph $\mathcal{E}$ as follows:
- Let $\{n_i\}_{i=0}^{N-1}$ denote the set of all e-nodes in the e-graph.
- Let $\{m_j\}_{j=0}^{M-1}$ denote the set of all e-classes, where the root e-class containing the top-level operator is indexed by 0.
- Each e-class m_j contains a set of e-nodes: $m_j \subseteq \{n_k\}_{i=0}^{N-1}$ and $|m_j|$ denotes the cardinality of this set.
- ch_i denotes the set of child e-classes for e-node of index i.
- pa_j denotes the set of parent e-nodes that depend on e-class of index j.
- $ec(i)$ returns the index of the e-class that contains e-node of index i, namely, $n_i \in m_{ec(i)}$.

Example: Formalizing an E-Graph

Formalism	Mapping to Example
E-Nodes (n_i)	$n_0 = \times$, $n_1 = \ll$, $n_2 = a$, $n_3 = 2$, $n_4 = 1$
E-Classes (m_j)	$m_0 = \{n_0, n_1\}$ (Root: $a \times 2 \equiv a \ll 1$) $m_1 = \{n_2\}$, $m_2 = \{n_3\}$, $m_3 = \{n_4\}$
Children (ch_i)	$ch_0(\text{for } \times) = \{m_1, m_2\}$ $ch_1(\text{for } \ll) = \{m_1, m_3\}$
Parents (pa_j)	$pa_1(\text{for } m_1) = \{n_0, n_1\}$ $pa_2(\text{for } m_2) = \{n_0\}$
Lookup $(ec(i))$	$ec(0) = 0, ec(1) = 0, ec(2) = 1, ec(3) = 2, ec(4) = 3.$



Section 2

Extraction

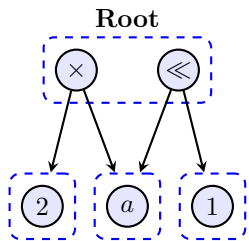
The Extraction Problem

- Equality Saturation yields a massive set of valid, equivalent programs.
- Eventually, the compiler must output a single, optimized AST to execute.
- This is the **Extraction Problem**: Given a cost function for each operation, find the tree with the absolute minimum cost.

Example:

Given Cost Function:

- Multiplication ($\times$): Cost $= 5$
- Bit Shift ($\ll$): Cost $= 1$
- Variables/Constants ($a, 1, 2$): Cost $= 0$



Exact Extraction: Integer Linear Programming (ILP)

- Let $n_i \in \{0, 1\}$ and $c_j \in \{0, 1\}$ indicate whether e-node i and e-class j are selected for the final program.
- **Cost Function (Objective):** Define a function $f : \mathbb{N} \to \mathbb{R}$ that defines the "cost" of selecting an enode.
  - This cost might represent some combination of performance, memory constraints, temperature constraints, etc.
- Then, our goal is to minimize $\sum_{i=0}^{N-1} f(i) \cdot n_i$ subject to the constraints that:
  - The root e-class must be selected. $(c_0 = 1)$
  - If an e-class is selected, exactly one of its e-nodes must be chosen to represent it. $(c_j = \sum_{n_k \in m_j} n_k \quad \forall j)$,
  - If an e-node is selected, *all* of its child e-classes must be selected to compute its arguments. $(n_i \leq c_k \quad \forall k \in ch_i, \forall i)$.

ILP

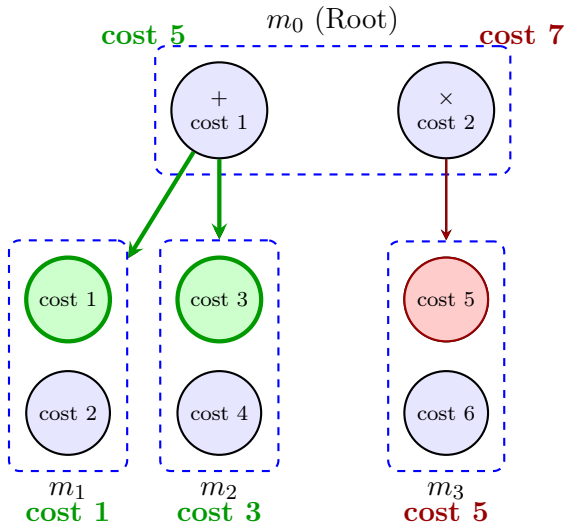
$$
\begin{aligned}
\text{Minimize} \quad & \sum_{i=0}^{N-1} f(i) \cdot n_i \\
\text{subject to} \quad & c_0 = 1 \\
& c_j = \sum_{n_k \in m_j} n_k && \forall j \\
& n_i \leq c_k && \forall k \in ch_i, \forall i \\
& n_i, c_j \in \{0, 1\} && \forall i, j
\end{aligned}
$$

- Can plug this into an ILP solver to solve in $\mathcal{O}(2^n)$ time.
- Problem: In defining the cost function, we are implicitly assuming *additive costs*, where each e-node's cost is independent of its children.
- Problem 2: Infeasible for most programs with a large number of nodes + rewrite combinations.

Greedy Extraction

- Because we assume additive costs, note that selecting each enode from an eclass only depends on its children.
- Thus, we can use a standard $O(n)$ reverse-topo sort dp to compute the min cost.
- Note the we assume the E-graph is a DAG.
- Is this assumption valid?
  - Consider the rule $A + B \to B + A$ or $A \to A \times 1$.
  - Useful to transform $A + (A \times B)$ to $A \times (1 + B)$ but creates a circular dependency.

Greedy DAG Extraction: An Example



Another approach: SmoothE

- One glaring issue with these approaches is that we can't always assume additive costs.
  - For example, the cost of an addition node decreases significantly if it has a multiplication node child (since hardware often groups both instructions in one cycle).

  SmoothE (ASPLOS 2025) solves this problem, while also providing an alternate way to optimize the cost function.

- Traditional ILP makes discrete, binary choices ($S \in \{0, 1\}$): pick node A or node B.
- SmoothE uses a **differentiable, probabilistic** extraction method:
  - Relaxes discrete decisions into a continuous probability vector ($\mathbf{p} \in [0, 1]^N$).
  - This transformation enables the use of modern Machine Learning optimizers, like Gradient Descent, over the E-graph directly on GPUs.

SmoothE Objective

$$\begin{aligned} &\text{minimize}_{\mathbf{cp}\in[0,1]^N} f(\mathbf{p}) \\ &\text{s.t.} \quad \mathbf{p} = \phi(\mathbf{cp}), \text{ acyclicity constraint} \end{aligned} \tag{1}$$

- $\mathbf{p} \in [0,1]^N$: probability vector for which e-nodes are chosen.
- $cp_i \in \mathbb{R}$: The conditional probability of choosing e-node i given that its e-class is selected. (These are the learned values)
  - $cp_i := P(n_i \text{ is chosen} \mid m_{ec(i)} \text{ is chosen}) = \frac{p_i}{P(m_{ec(i)} \text{ is chosen})}$
  - $cp_k \in [0,1], \quad \sum_{n_k \in m_{ec(i)}} cp_k = 1$
- $f(\mathbf{p})$: $\mathbb{R}^N \to \mathbb{R}$: cost function of some probability assignment $\mathbf{p}$
- $\phi(\mathbf{cp})$: $\mathbb{R}^N \to \mathbb{R}^N$: extraction function mapping conditional probabilities $\mathbf{cp}$ to selection probabilities $\mathbf{p}$. (computed from $\mathbf{cp}$).

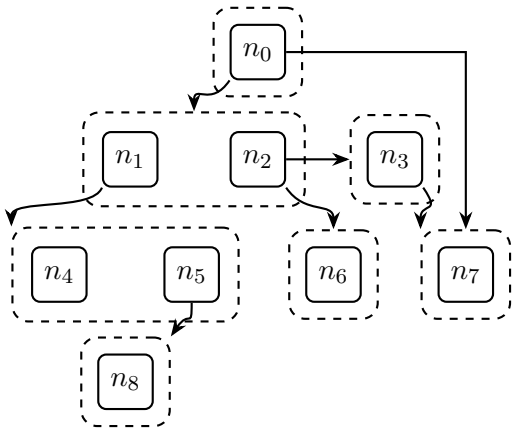
The function ϕ

- We assume that for every enode, the probability a given parent node is selected is independent.
- Then by independence,

$$P(m_{ec(i)} \text{ is chosen}) = 1 - \prod_{n_k \in pa_{ec(i)}} P(n_k \text{ is not chosen}) = 1 - \prod_{n_k \in pa_{ec(i)}} (1 - p_k),$$

  and $p_i = \phi(cp_i) = cp_i \cdot P(m_{ec(i)} \text{ is chosen})$.
- If the graph is a DAG, we can compute these probabilities in topological order (starting with $p_0 = 1$).
- Otherwise, we update the probabilities for every node in parallel, until all probabilities converge.
  - Because the probability updates are monotonic increasing and strictly bounded in $[0, 1]$, this is a *fixed-point iteration* that is guaranteed to converge.

Example Computation Diagram



n	cp_n	p_n
0	1	1
1	cp_1	cp_1
2	$1 - cp_1$	$1 - cp_1$
3	1	$1 - cp_1$
4	cp_4	$cp_1 cp_4$
5	$1 - cp_4$	$cp_1 - cp_1 cp_4$
6	1	$1 - cp_1$
7	1	1
8	1	$cp_1 - cp_1 cp_4$

What about the Acyclicity Constraint?

Theorem (NOTEARS - formulation): Let $\bar{Z}$ be the transition matrix for an unweighted directed graph with n vertices. The graph is acyclic if and only if: $Q(\bar{Z}) = \text{tr}(e^{\bar{Z}}) - n = 0$.

Proof Intuition:

- $\bar{Z}^g[r, c]$ represents the number of g-hop paths from the r-th node to the c-th node.
- Since $\bar{Z}$ is non-negative, $\text{tr}(\bar{Z}^g) = 0$ iff there are no g-hop cycles.
- Using the Taylor expansion for the matrix exponential:

$$\text{tr}(e^{\bar{Z}}) = \text{tr}(I) + \text{tr}(\bar{Z}) + \frac{1}{2!}\text{tr}(\bar{Z}^2) + \cdots \geq n$$

- The equality ($\text{tr}(e^{\bar{Z}}) = n$) is attained if and only if the graph has no cycles of any length.

Handling Acyclicity in SmoothE

- In our case, the Transition Matrix $\bar{Z}$ is the sum of the conditional probabilities of all E-nodes n_k in E-class m_i which have E-class m_j as a child.

$$Z_t[i,j] = \sum_{k \text{ s.t. } n_k \in m_i \text{ and } m_j \in ch_k} cp_k$$

- This measures the probability if E-class m_i is selected, the compiler is forced to also pick E-class m_j..
- The NOTEARS term $Q(\bar{Z}) = \text{tr}(e^{\bar{Z}}) - n$ is added as a penalty to the main objective, with lagrange multipler λ.

Refined SmoothE Optimization:

$$\begin{aligned} &\text{minimize}_{\mathbf{cp} \in [\mathbf{0},\mathbf{1}]^{\mathbf{N}}} \quad \mathcal{L} = f(\mathbf{p}) + \lambda \cdot Q(\bar{Z}) \\ &\text{s.t.} \quad \mathbf{p} = \phi(\mathbf{cp}) \end{aligned} \tag{2}$$

SmoothE Results

Table 2. Comparative results using the linear cost model across 5 realistic datasets — Increases in solution quality are normalized to the oracle obtained by running CPLEX for 10 hours. The time limit is set as 15 minutes for all ILP methods. In each dataset, **time** is reported in second. **worst** indicates the statistic of the worst performing e-graph, and **avg.** reports the geometric mean across e-graphs with feasible solutions. The red number in parentheses indicates the number of e-graphs for which the solver fails to provide a valid solution. The maximum difference for SmoothE is based on 3 runs. For diospyros, flexc, impress, rover, and tensat, we use independent, correlated, correlated, independent, and independent assumption respectively.

Dataset	CPLEX ILP time (fails) worst / avg.	SCIP ILP time (fails) worst / avg.	CBC ILP time (fails) worst / avg.	Heuristic (egg) time (fails) worst / avg.	Heuristic+[22] time (fails) worst / avg.	SmoothE (ours) time (fails) worst / avg.
diospyros	211.8 19.6% / 1.4%	240.5 (1) Failed / 7.5%	350.4 (1) Failed / 2.5%	0.3 0.0% / 0.1%	0.2 0.0% / 0.1%	$8.4_{\pm 0.7}$ $0.1\%_{\pm 0.0\%}$ / $0.1\%_{\pm 0.0\%}$
flexc	5.2 0.0% / 0.0%	41.6 0.0% / 0.0%	585.4 (5) Failed / 66.6%	0.0 2.8% / 1.2%	0.1 2.8% / 1.2%	$19.3_{\pm 1.7}$ $0.7\%_{\pm 0.0\%}$ / $0.2\%_{\pm 0.0\%}$
impress	39.5 0.0% / 0.0%	69.8 0.0% / 0.0%	384.5 220% / 30.8%	0.4 280% / 53.0%	1.0 0.0% / 0.0%	$4.6_{\pm 0.0}$ $0.0\%_{\pm 0.0\%}$ / $0.0\%_{\pm 0.0\%}$
rover	520.4 4.3% / 0.5%	671.8 45.6% / 17.2%	677.6 58.6% / 19.6%	0.1 11.0% / 2.9%	0.1 11.0% / 2.9%	$20.6_{\pm 1.3}$ $4.4\%_{\pm 1.2\%}$ / $0.2\%_{\pm 0.1\%}$
tensat	678.2 9.7% / 1.9%	900.0 (1) Failed / 260%	900.0 (1) Failed / 67.2%	0.4 46.4% / 12.1%	1.3 46.4% / 11.9%	$24.4_{\pm 2.5}$ $17.0\%_{\pm 0.0\%}$ / $4.6\%_{\pm 0.1\%}$

Section 3

Applications

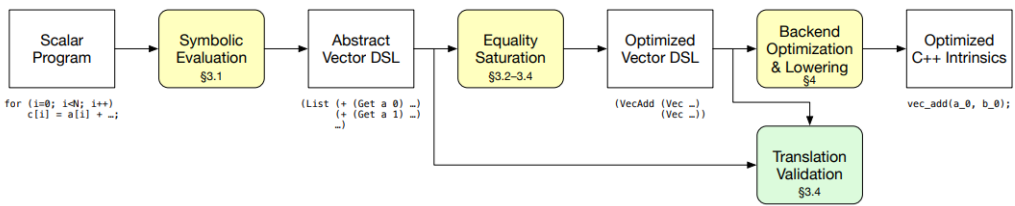
The egg Framework

- Equality Saturation has seen an explosion of adoption, largely driven by egg **(POPL 2021)**.
- `egg` is a fast, robust, and extensible E-graph library built in Rust.
- The Developer API consists of defining a language, rewrite rules, analysis, Runner, and Extractor.
- This open-source framework has lowered the barrier for many applications to utilize E-graphs for optimization.

Application: Diospyros

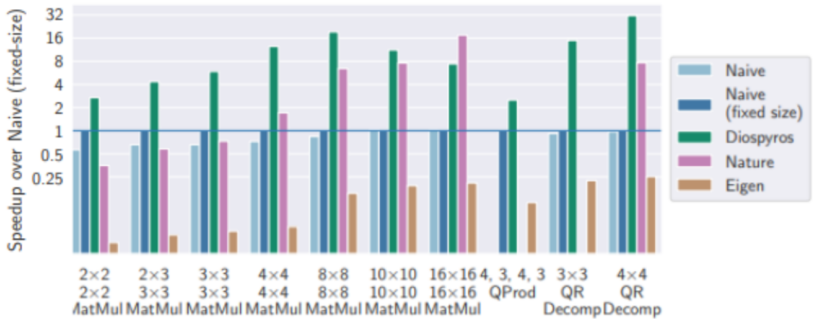
- **Diospyros (ASPLOS 2021)** is an optimizing compiler for Digital Signal Processing (DSP) architectures.
- DSP chips contain highly specialized vector instructions, often used when running ML training/inference.
- Traditional compiler heuristics struggle to map standard code to these instructions.
- Diospyros uses E-graphs to explore complex algebraic rearrangements.
- By representing vectorizations as rewrite rules, it discovers highly efficient SIMD usage that often outperforms hand-tuned C code.

Diospyros Translation Diagram



Diospyros Results

- Naive is a generic loop nest, Naive (fixed size) is a loop nest with fixed bounds.
- Nature is a vendor-supplied library function, and Eigen is a C++ template linear algebra library.



Application: Floating-Point Optimization (Herbie)

- **Herbie (PLDI 2015)** automatically improves the accuracy of floating-point expressions.
- Because floating-point math is not associative, $(a + b) + c \neq a + (b + c)$. Finding the most numerically stable arrangement is difficult.
- By translating expressions into an E-graph, Herbie explores all valid mathematical reformulations using real-number axioms.

How Herbie Works:

- For the inputs with high error, Herbie then *localizes* the error by computing the error for each operation in the formula $\frac{-b+\sqrt{b^2-4ac}}{2a}$.
- Herbie contains hundreds of algebraic rewrites in a rewrite table that it tries to use over several iterations.
  - For example, this may rewrite $x + y \to \frac{x^2-y^2}{x-y}$ to prevent cancellation when $b^2 \gg 4ac$, to obtain

$$\frac{(-b)^2 - \left(\sqrt{b^2-4ac}\right)^2}{(-b) - \sqrt{b^2-4ac}}/2a = \frac{-2c}{b + \sqrt{b^2-4ac}}$$

- Algebraic rewrites cannot always fix errors near zero or infinity, so Herbie also generates polynomial approximations (like Taylor series) to handle extreme input values.
  - For example, this may rewrite $\sqrt{1-x} \approx 1 - \frac{x}{2}$ to prevent cancellation when $b^2 \gg 4ac$, to obtain:

How Herbie Works: Taylor Expansion

$$\frac{-b + b\sqrt{1 - \frac{4ac}{b^2}}}{2a} \approx \frac{-b + b\left(1 - \frac{2ac}{b^2}\right)}{2a} = \frac{-b + b - \frac{2ac}{b}}{2a} = -\frac{c}{b}$$

- After generating all candidate expressions, Herbie uses a DP algorithm to compute the best possible expression for each input regime (similar to Segmented Least Squares).
  - You don't always want to take the minimum for every data point since this is computationally expensive!
- The final output is a numerically stable piecewise function:

$$f(a,b,c) = \begin{cases} -\frac{c}{b} & \text{if } b^2 \gg 4ac \text{ (Extreme cancellation)} \\ \frac{-2c}{b+\sqrt{b^2-4ac}} & \text{if } b > 0 \text{ (Standard cancellation)} \\ \frac{-b+\sqrt{b^2-4ac}}{2a} & \text{otherwise} \end{cases}$$

Herbie Evaluation

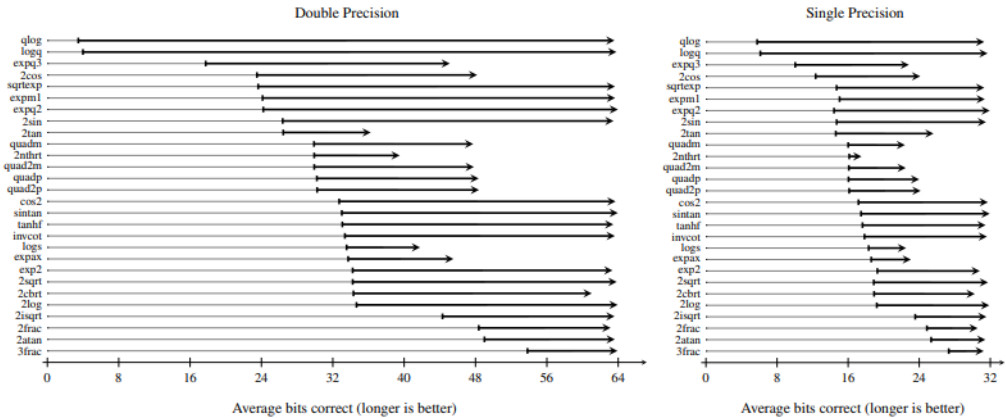


Figure 7. Each row represents the improvement in accuracy achieved by Herbie on a single benchmark. The thick arrow starts at the accuracy of the input program, and ends at the accuracy of Herbie's output. Accuracy is measured by the number of correct output bits, averaged across 100 000 random input points.

Future Directions in Equality Saturation

- **Machine Learning for Extraction:**
  - Replacing expensive ILP solvers with neural networks to learn heuristic cost models and guide the extraction process (ex. SmoothE).
- **Relational E-Matching (`egglog`):**
  - Combining the framework of Datalog with E-graphs to achieve dramatically faster, database-style query execution for pattern matching.

Conclusion

- E-graphs fundamentally shift compiler design from *destructive rewriting* to *constructive exploration*.
- While extraction remains a difficult NP-hard bottleneck, innovations like SmoothE push the boundaries of what is scalable.
- Through open-source frameworks like `egg`, this paradigm is revolutionizing how programs are optimized across many industries.
- Thank you! Any questions?

References

- **egg:** Willsey et al. *egg: Fast and Extensible Equality Saturation.* POPL 2021.
- **Diospyros:** Nandi et al. *Vectorization for Digital Signal Processors via Equality Saturation.* ASPLOS 2021.
- **Herbie:** Panchekha et al. *Automatically Improving Accuracy for Floating Point Expressions.* PLDI 2015.
- **SmoothE:** Phothilimthana et al. *SmoothE: Differentiable E-Graph Extraction.* ASPLOS 2025.
- **Foundations:** Tate et al. *Equality Saturation: A New Approach to Optimization.* POPL 2009.

Brainteaser

The evil Dr. Franklin Zhang has captured n innocent CS 374 students numbered 1 through n. The students are forced into a building with $n+1$ rooms: n individual prison cells and Franklin's great evil throne room.

After a few days, Franklin gets bored, so he decides to call up his prisoners to entertain him. For all of eternity, he calls his prisoners one at a time to his room, but no one else knows that a prisoner is being called. Also, Franklin likes all his prisoners, so all of his prisoners will be called infintely many times.

Brainteaser

In the center of Franklin's room, Franklin keeps his fine dining glass. This glass has a special property: it can balance upside-down! When a CS 374 student comes up to his room, they have the option to flip the glass or keep it the way it is currently. They may not convey any other information through the glass or in the room.

At any point, any student can press a special button in their room when they believe that every student has met with Dr. Franklin. If they are right, all the students are free; if not, they will become the Greedy Dragon's next snack!

Initially, the glass is face up, and everyone knows this. Dr. Franklin, the generous man that he is, lets the students devise a strategy before they are confined to their cells. Can the students escape eventually, or are they forever doomed to Dr. Franklin's evil lair?