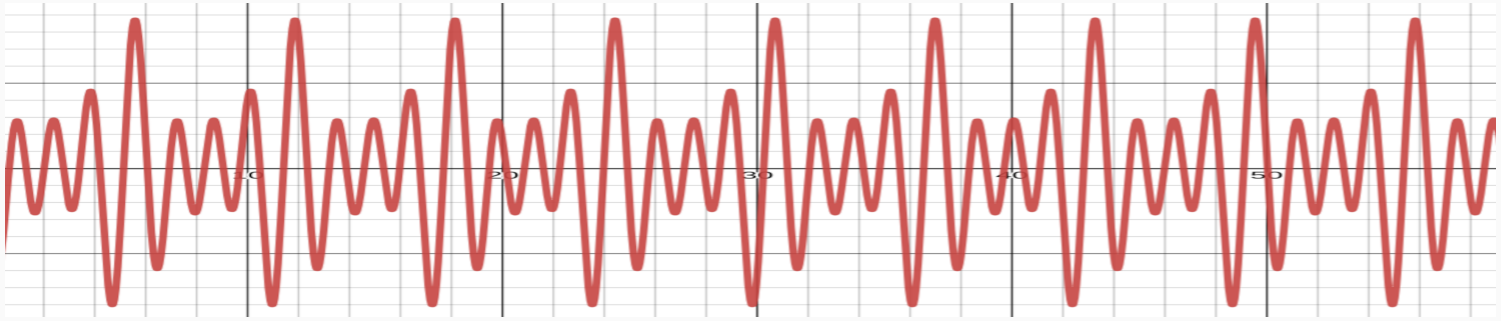
Model Simplification



Prof. Fabrício Olivetti de França

Federal University of ABC

15 June, 2024

Simplifying Algebraic Expressions

Simplifying Algebraic Expressions

Rewriting an algebraic expression by applying equivalence rules to generate an equivalent expression that is more efficient and compact.

Equivalent Expressions

Equivalent expressions are those that are equal in value, even if they look different.

Equivalent Expressions

$$x + y \equiv y + x$$

$$2x \equiv x + x$$

$$\tanh x \equiv \frac{e^{2x} - 1}{e^{2x} + 1}$$

Efficiency and Compactness

There is no standard measure of efficiency and compactness for an algebraic expression as this depends on the application:

- Smaller expressions
- Computationally more efficient
- Less repeated use of the same variable
- Less numerical constants
- etc.

Efficiency and Compactness

For example:

- $x + x + x$ could be much faster to calculate than $3x$ in older architectures.
- $3x$ is more precise if we are using interval arithmetic.
- $(x + 1.5y)/(1 + 3x)$ can be more interpretable than $(2x + 3y)/(2 + 6x)$ (if these parameters can still fit the data).

Equivalence Rules

This simplification process depends on a set of equivalence rules that describes whether two expressions are equivalent. Let's see a couple more examples:

$$x + y \equiv y + x$$

$$xy \equiv yx$$

$$2x \equiv x + x$$

$$x + (y + z) \equiv (x + y) + z$$

$$x(y + z) \equiv xy + xz$$

$$(xy)/z \equiv x(y/z)$$

Optimal Expression

Finding the optimal expression is not as simple as applying these rules sequentially. The order of the application matters, enumerating all possible orders leads to a combinatorial explosion.

If we have N rules and we can only apply them once, we have $N!$ ways of applying these rules.

Of course, not every rule is appliable at every moment and we may need to apply a rule more than once.

This is a well known problem in compiler optimization called **Phase Ordering Problem**:

> Given a measure of goodness for each rule at a given expression, if two rules give the same local benefit, then both can lead to sub-optimal results.

Program Optimization

Let's see this problem using algebraic expressions $(2x)/2$ and the following rules:

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

Program Optimization

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$\alpha = x$$

$$(2x)/2 = (x << 1)/2$$

Program Optimization

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$(2x)/2 = (x << 1)/2$$

Program Optimization

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$(2x)/2 = (x << 1)/2$$

Program Optimization

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$(2x)/2 = (x << 1)/2$$

Program Optimization

$$2\alpha \equiv \alpha << 1$$

$$\alpha\beta \equiv \beta\alpha$$

$$1\alpha \equiv \alpha$$

$$\alpha/\alpha \equiv 1$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$(2x)/2 = (x << 1)/2$$

Program Optimization

That was quick and sub-optimal! Let's try again:

$$\alpha\beta \equiv \beta\alpha$$
$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$
$$\alpha/\alpha \equiv 1$$
$$1\alpha \equiv \alpha$$
$$2\alpha \equiv \alpha << 1$$

$$\alpha = 2$$
$$\beta = x$$

$$(2x)/2 = (x2)/2$$

Program Optimization

$$\alpha\beta \equiv \beta\alpha$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$\alpha/\alpha \equiv 1$$

$$1\alpha \equiv \alpha$$

$$2\alpha \equiv \alpha << 1$$

$$\alpha = x$$

$$\beta = 2$$

$$\gamma = 2$$

$$(2x)/2 = (x2)/2$$

$$(x2)/2 = x(2/2)$$

Program Optimization

$$\alpha\beta \equiv \beta\alpha$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$\alpha/\alpha \equiv 1$$

$$1\alpha \equiv \alpha$$

$$2\alpha \equiv \alpha << 1$$

$$\alpha = 2$$

$$(2x)/2 = (x2)/2$$

$$(x2)/2 = x(2/2)$$

$$x(2/2) = x1$$

Program Optimization

$$\alpha\beta \equiv \beta\alpha$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$\alpha/\alpha \equiv 1$$

$$1\alpha \equiv \alpha$$

$$2\alpha \equiv \alpha << 1$$

$$(2x)/2 = (x2)/2$$

$$(x2)/2 = x(2/2)$$

$$x(2/2) = x1$$

Program Optimization

So close! But if we go through the rules again we will reach the optimal result!

$$\alpha\beta \equiv \beta\alpha$$
$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$
$$\alpha/\alpha \equiv 1$$
$$1\alpha \equiv \alpha$$
$$2\alpha \equiv \alpha << 1$$

$$(2x)/2 = (x2)/2$$
$$(x2)/2 = x(2/2)$$
$$x(2/2) = x1$$

Program Optimization

$$\alpha\beta \equiv \beta\alpha$$

$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$

$$\alpha/\alpha \equiv 1$$

$$1\alpha \equiv \alpha$$

$$2\alpha \equiv \alpha << 1$$

$$\alpha = x$$

$$\beta = 1$$

$$(2x)/2 = (x2)/2$$

$$(x2)/2 = x(2/2)$$

$$x(2/2) = x1$$

$$x1 = 1x$$

Program Optimization

Good!

$$\alpha\beta \equiv \beta\alpha$$
$$(\alpha\beta)/\gamma \equiv \alpha(\beta/\gamma)$$
$$\alpha/\alpha \equiv 1$$
$$1\alpha \equiv \alpha$$
$$2\alpha \equiv \alpha << 1$$

$$\alpha = x$$

$$(2x)/2 = (x2)/2$$
$$(x2)/2 = x(2/2)$$
$$x(2/2) = x1$$
$$x1 = 1x$$
$$1x = x$$

Destroy the Past!

One reason for the phase ordering problem is that the program transformation is destructive.

Whenever we apply a rule, we *forget* about the past equivalent expressions. So, whenever we take the wrong *path*, we cannot go back.

Equality Saturation

Non-destructive rewriting

In [1] Tate introduced the idea of **Equality Saturation** proposing a method for non-destructive rewriting of the original expression.

The main idea is that all the rules are applied in parallel and the resulting expressions are kept in a compact data structure now known as **e-graph**.

[1] Tate, R., et al, “Equality Saturation: a New Approach to Optimization,” in Logical Methods in Computer Science, 2011.

Non-destructive rewriting

This technique became popular after a fast and customizable implementation called *egg*[2]. Egg is implemented in Rust and enabled the user to experiment with equality saturation in different applications with the requirements of:

- Defining a Language
- Writing the rules

[2]Willsey, M., et al. “egg: Fast and extensible equality saturation,” in Proceedings of the ACM on Programming Languages, vol. 5, no. POPL, pp. 1–29, 2021

Equality Saturation

An e-graph[3] is composed of:

- a set of e-classes (dashed) with a non-empty sets of e-nodes (solid line).
- a set of e-nodes representing a symbol of our language
- a set of edges connecting e-nodes to e-classes.

[3] Nelson, Charles Gregory. Techniques for program verification. Stanford University, 1980.

Equality Saturation

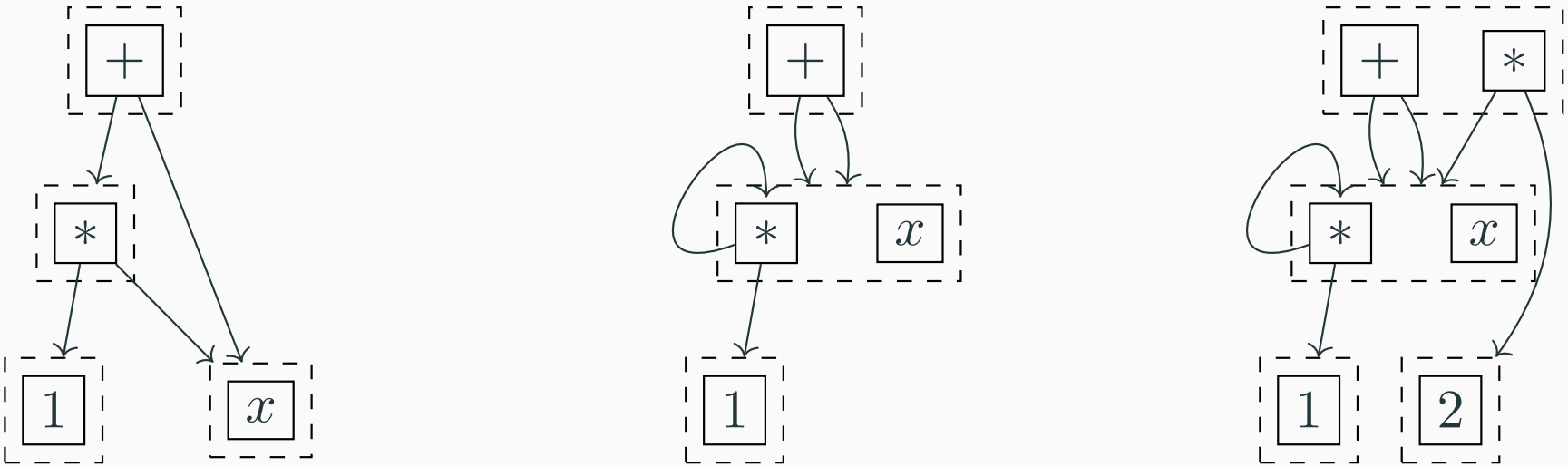
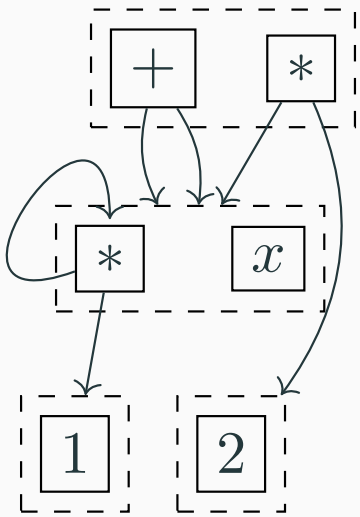


Figure 1: (a) Original expression; (b) after applying the rule $1 * x \rightarrow x$; (c) after applying the rule $x + x \rightarrow 2 * x$.

Equality Saturation

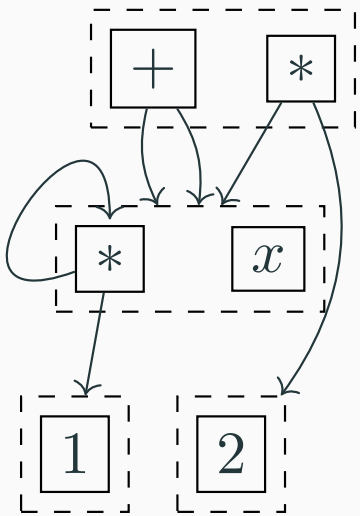
Notice how the children of an e-node points to the e-classes that contain one or more equivalent sub-expression. For example, we can go from $+$ to $1x$ or x or $1 \times 1 \times x$.



Equality Saturation

At a certain point, whenever we apply the rules the graph will not change! This means we reached the fixed point and the graph is saturated.

The saturated graph represents every equivalent expression reachable by the specific set of rules.



egg: Fast and Extensible Equality Saturation

egg in a Nutshell

```
def equality_saturation(expr, rewrites):
    egraph = build_egraph(expr)
    while !egraph.is_saturated_or_timeout():
        matches = []

        # read-only
        for rw in rewrites:
            for (subst, eclass) in egraph.search(rw.lhs):
                matches.append((rw, subst, eclass))

        # write-only
        for (rw, subst, eclass) in matches:
            new_eclass = egraph.add(rw.rhs.subst(subst))
            egraph.union(eclass, new_eclass)

        # restore invariants: merge congruent e-classes
        egraph.rebuild()

    return egraph.extract_best()
```

Equality Saturation

In the following, let's use as an illustrative example the expression $(2/x)(x+x)$ and the rules:

$$\alpha + \alpha \to 2\alpha$$
$$(\alpha/\beta)\gamma \to \alpha(\gamma/\beta)$$
$$(\alpha\beta)/\gamma \to \alpha(\beta/\gamma)$$
$$\alpha/\alpha \to 1$$
$$\alpha \cdot 1 \to \alpha$$

e-node representation

The representation of an e-node follows the fixed point of the representation of the program or expression.

In mathematics, a **fixed point** x of a function f, also known as **invariant point** is a value x such that $f(x) = x$. The **least fixed point** is the smallest fixed point x among all fixed points of a function.

e-node representation

As a data structure we can represent a fixed point as:

```haskell
data Fix f = Fix (f (Fix f))
```

which allows us to implement generic algorithms for folding and unfolding data structures.

e-node representation

A natural way to represent an expression tree is using a recursive type:

```
1  data Expr   = Const Double
2              | Var String
3              | Add Expr Expr
4              | Sub Expr Expr
5              | Mul Expr Expr
6              | Div Expr Expr
```

e-node representation

We can represent the same structure as its least fixed point with:

```haskell
data Expr a = Const Double
            | Var String
            | Add a a
            | Sub a a
            | Mul a a
            | Div a a

type FixExpr = Fix Expr
```

Notice that this structure allows a recursive pattern by replacing the type parameter `a` with `FixExpr`.

e-node representation

This structure also give us many conveniences, we can represent and expression `x + 2` as:

```
Fix (Add (Fix (Var "x") (Fix (Const 2))) :: FixExpr
```

which types check. But also, we can have represent the e-node `Add` point to the e-classes `3` and `1`:

```
Add 3 1 :: Expr Int
```

or a representation of the pattern `_ + _` where each `_` matches anything:

```
Add () () :: Expr ()
```

e-node representation

In dynamically typed language this can be expressed as structures with branches that can assume different types as it seems fit.

```
class Add:
    left: Any
    right: Any

class Mul:
    left: Any
    right: Any

class Var:
    name: String

class Const:
    val: float

Expr = Add | Mul | Var | Const
```

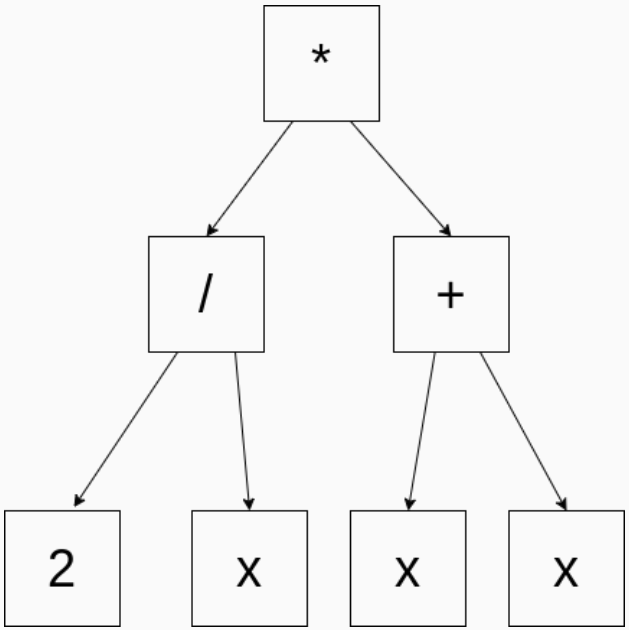
e-node representation

This definition allows us to represent our expression as an expression tree where each children of a node is another tree or an integer representing the e-class id it points to or a bottom symbol.

We will see next how this flexibility is useful for the implementation of egg.

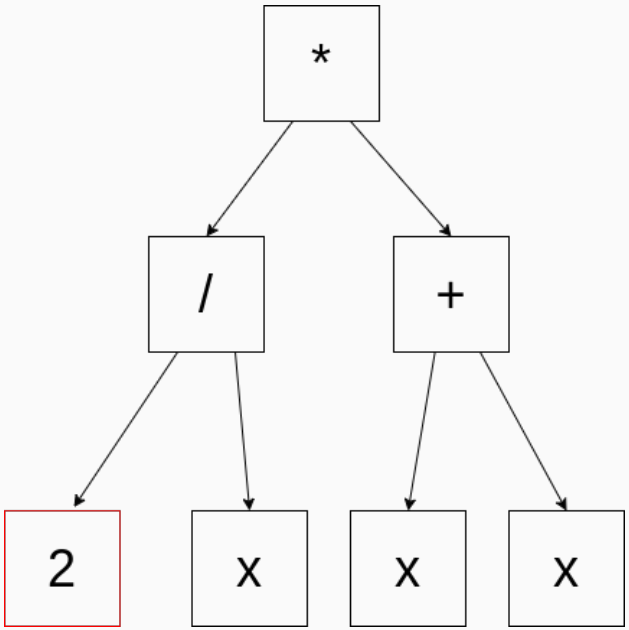
Expression to e-node

Starting with our exression in a tree-like representation:



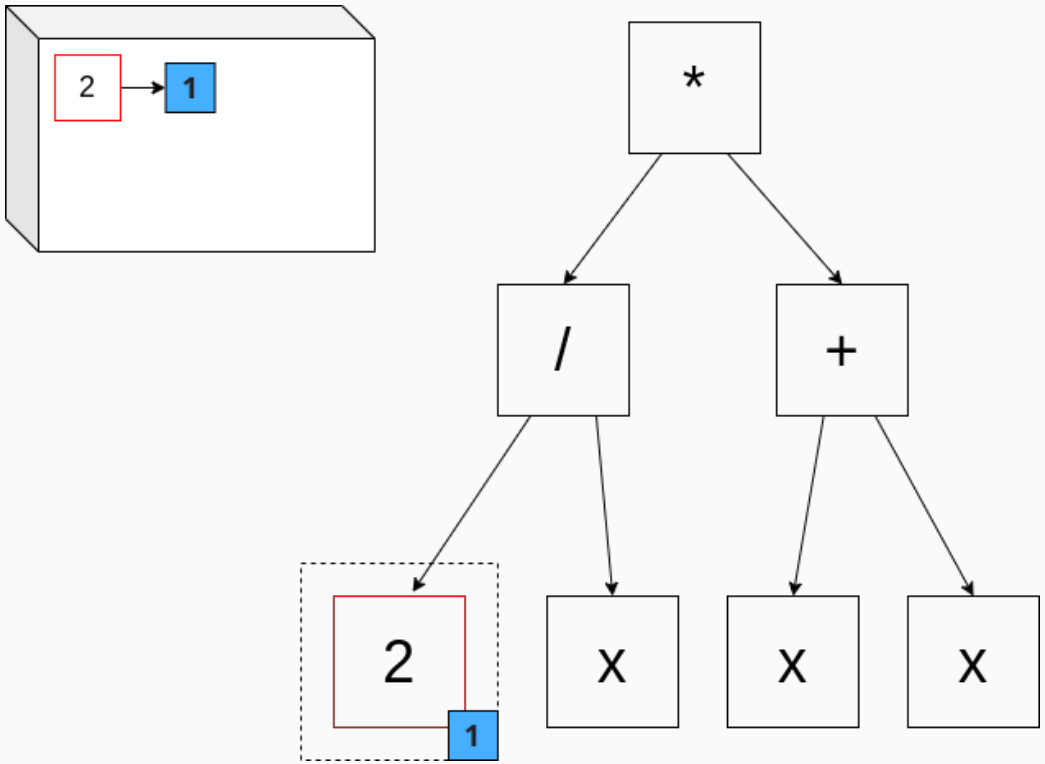
Expression to e-node

Visiting the nodes in a post-order traversal…



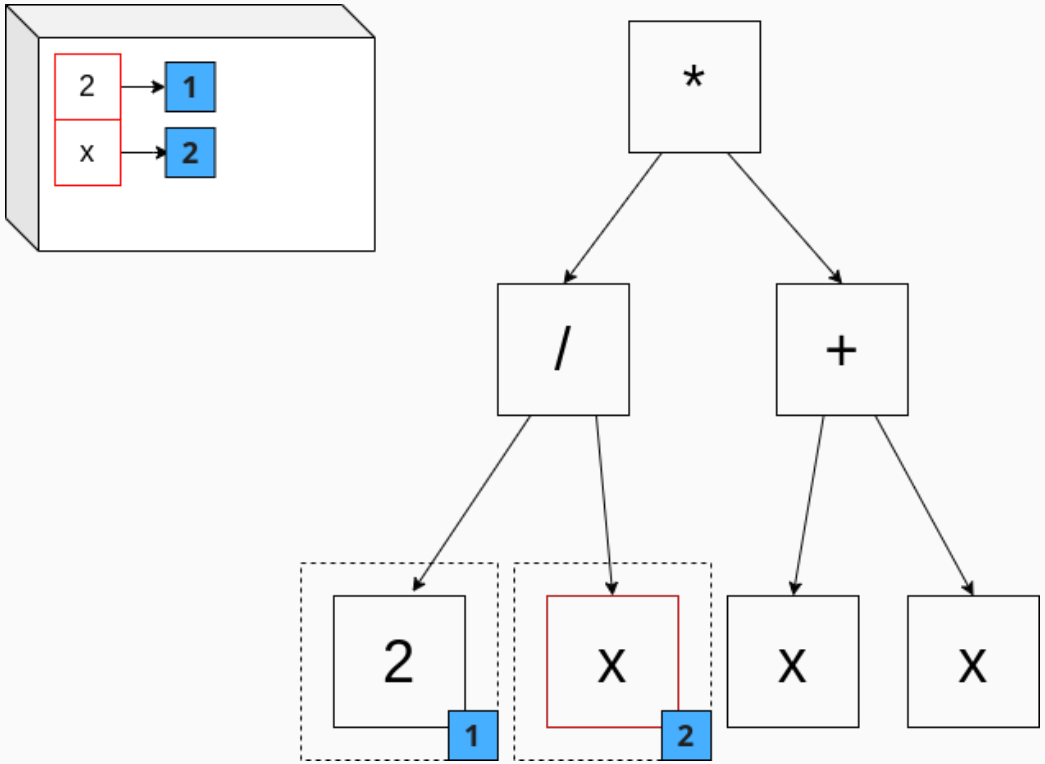
Expression to e-node

We create a new e-node assiging it a new e-class id. This information is stored in a hashcon, where the key is the node and the value is the corresponding e-class id.



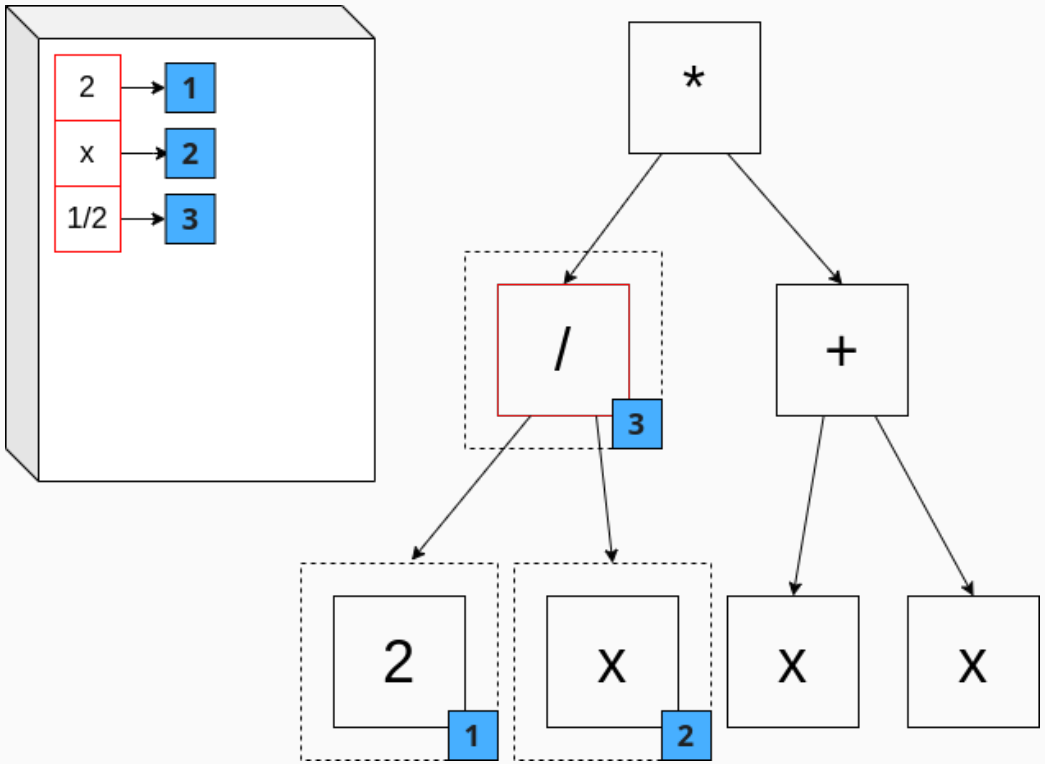
Expression to e-node

We create an e-node storing it into a new e-class and assign an e-class id.



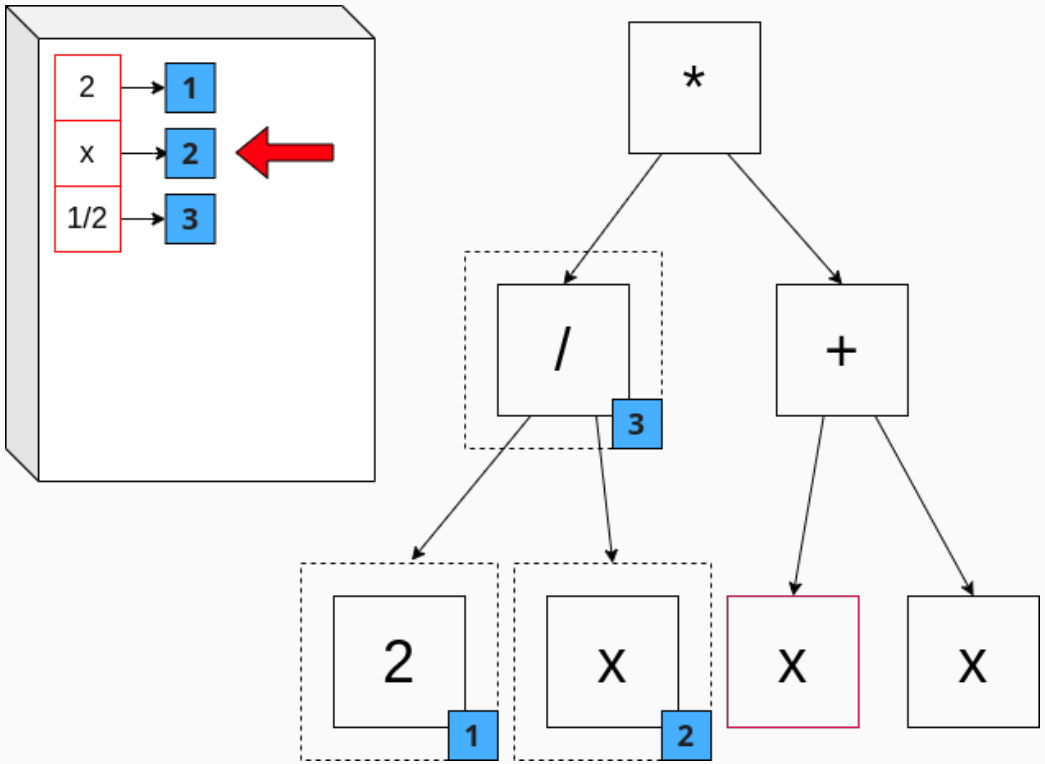
Expression to e-node

Any internal node will have their children replace by the corresponding e-class ids. We keep a hash table of this node representation to the corresponding assigned id.



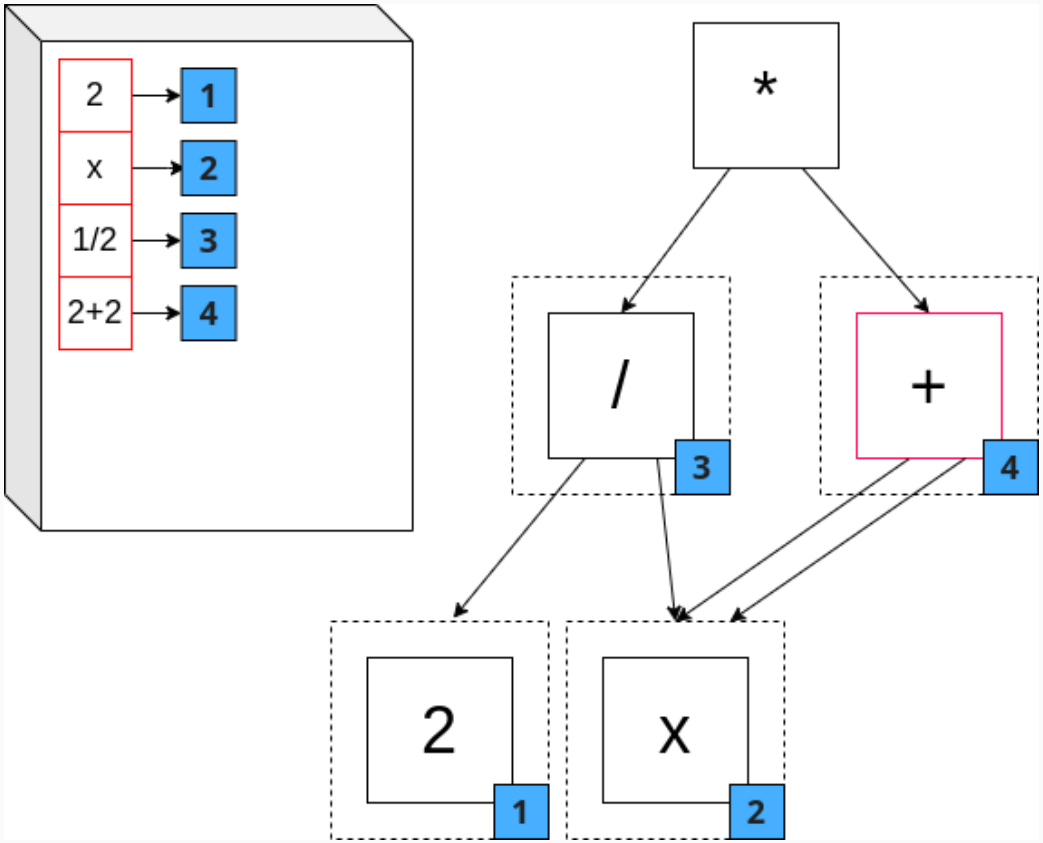
Expression to e-node

Whenever we reach a node contained in the hash table, we just retrieve the e-class id it belongs to.



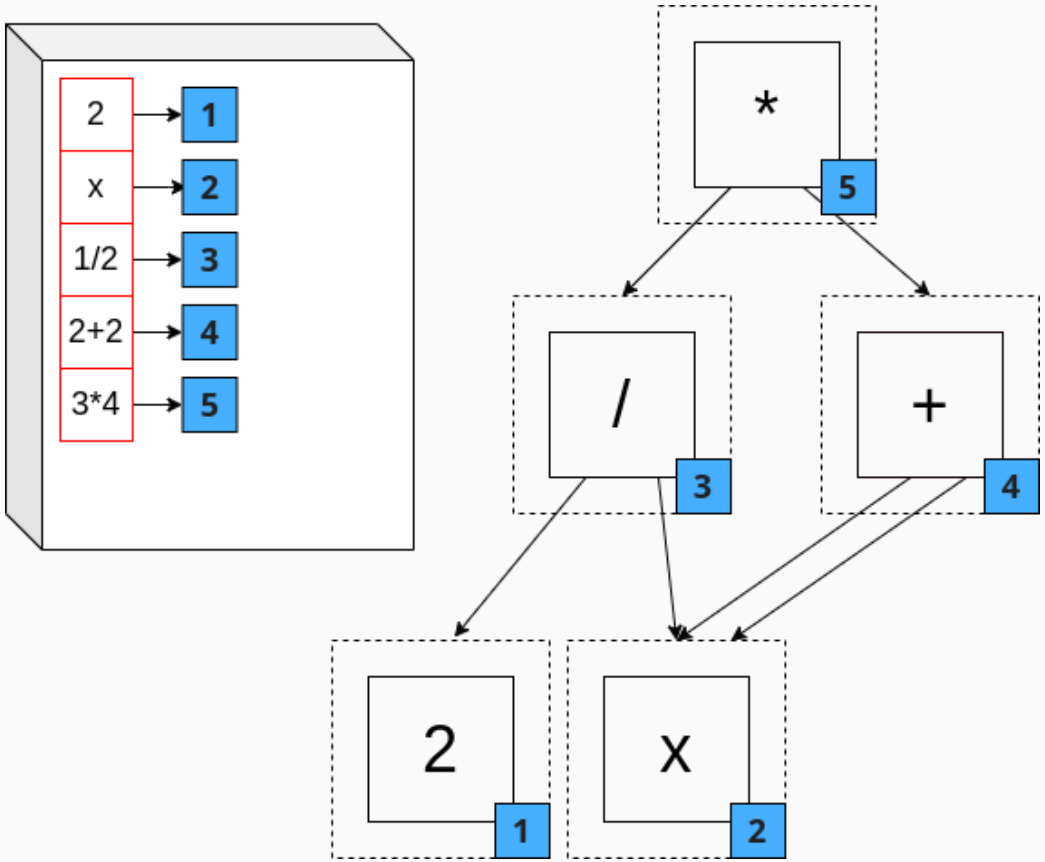
Expression to e-node

This will ensure that the graph is compact and the equivalent nodes are grouped together.



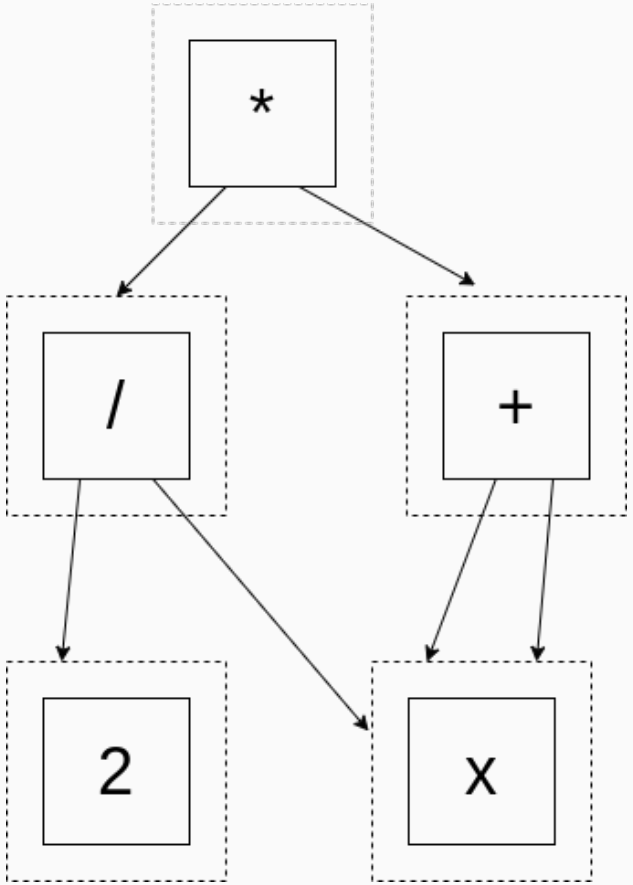
Expression to e-node

With this representation, it becomes easier to search the patterns for a match.



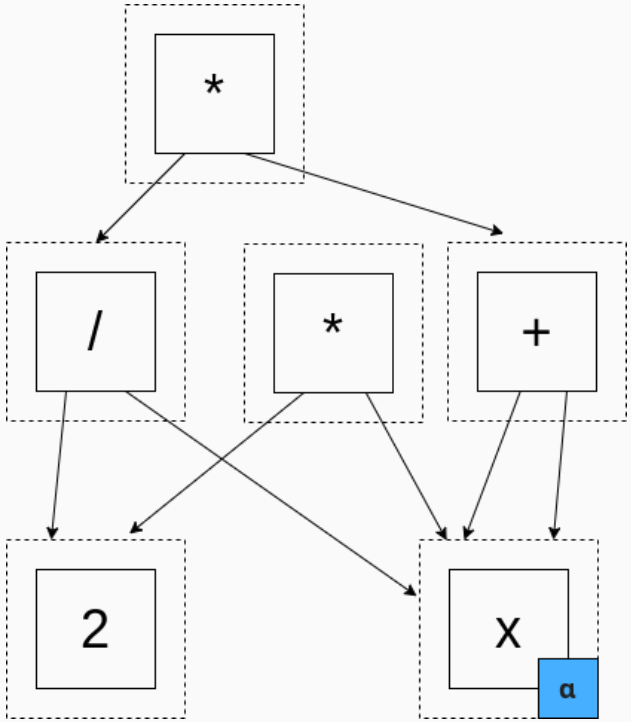
Equality Saturation

The corresponding e-graph of the expression $(2/x)(x+x)$ is



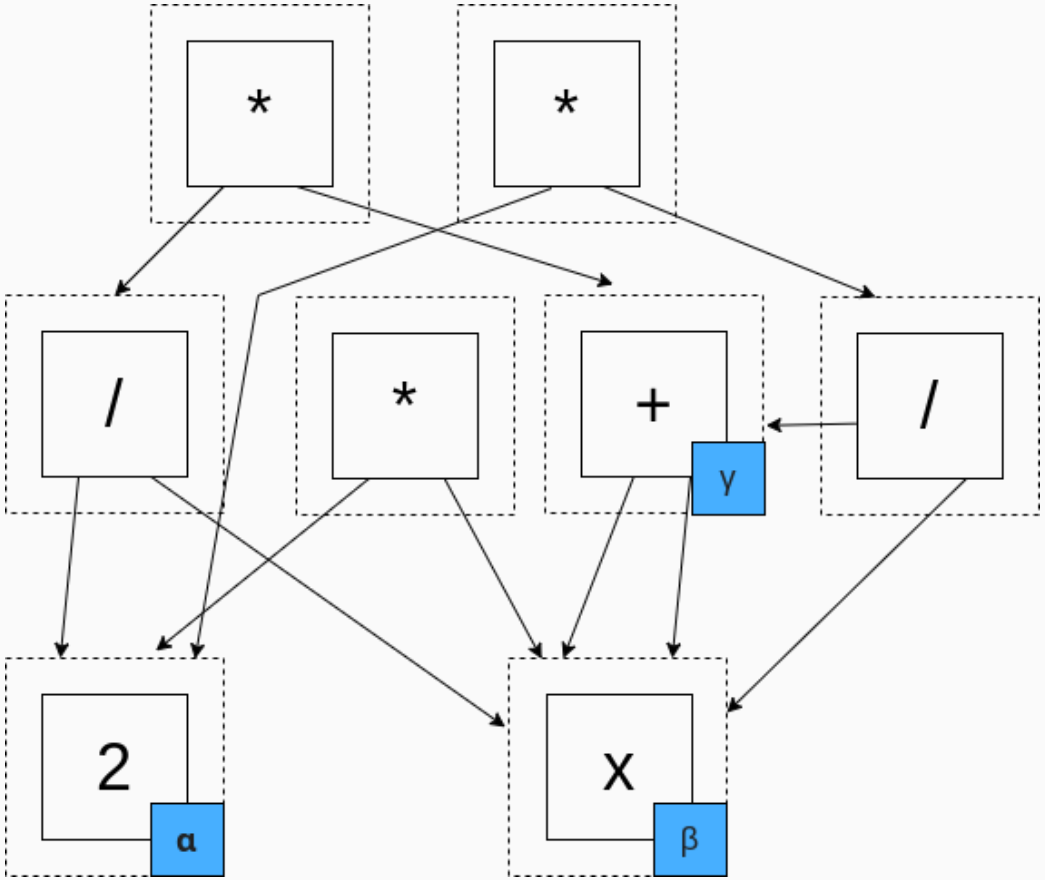
Equality Saturation

By applying the rule $x + x \rightarrow 2x$ we get



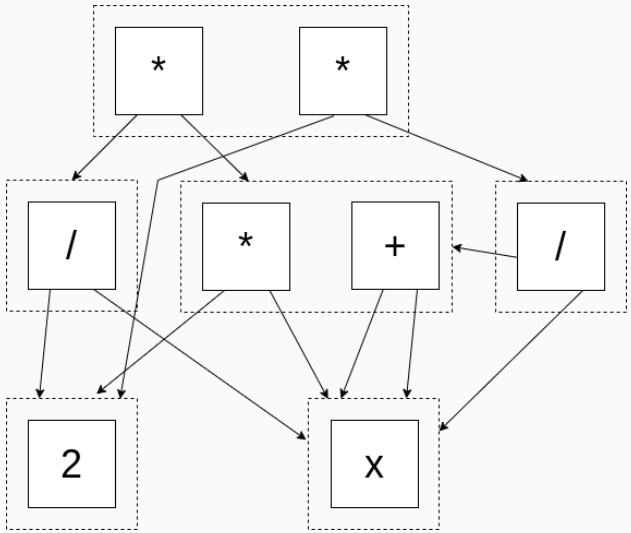
Equality Saturation

By applying the rule $(x/y)z \rightarrow x(z/y)$ we get



Equality Saturation

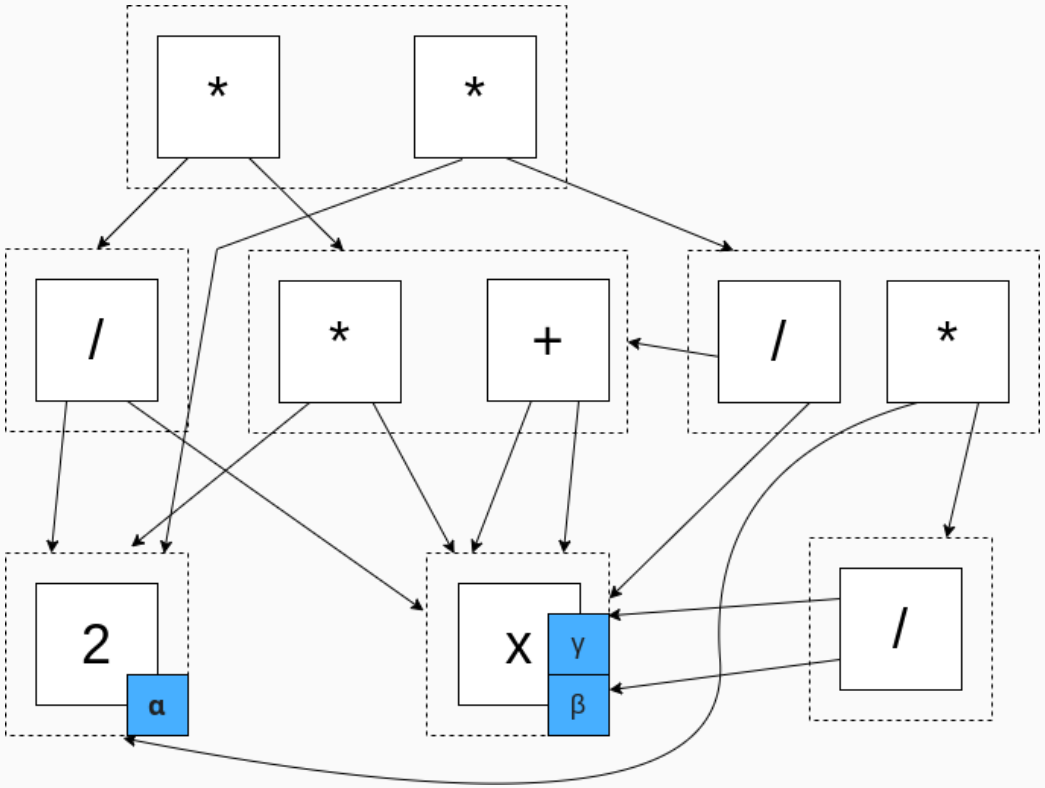
After this iteration, we must merge the e-nodes that are equivalent to the same e-class:



Notice how following any path from a given e-class will return equivalent expressions.

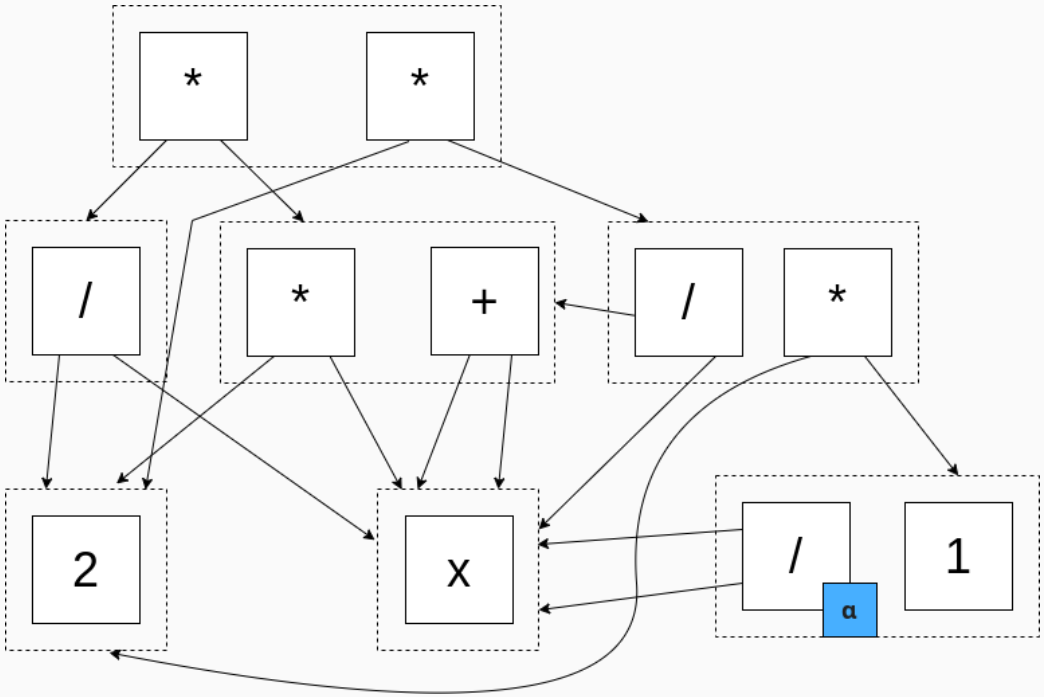
Equality Saturation

After applying every rule, we start again ignoring the already performed rewritings. Now we can match $(xy)/z \to x(y/z)$:



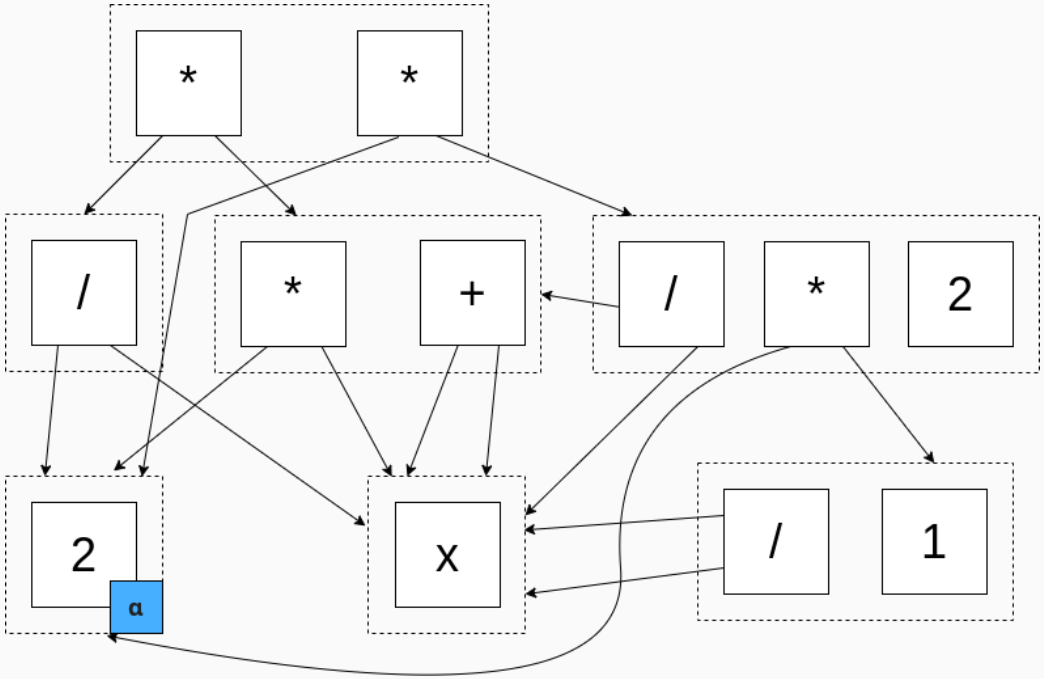
Equality Saturation

And now $x/x \rightarrow 1$:



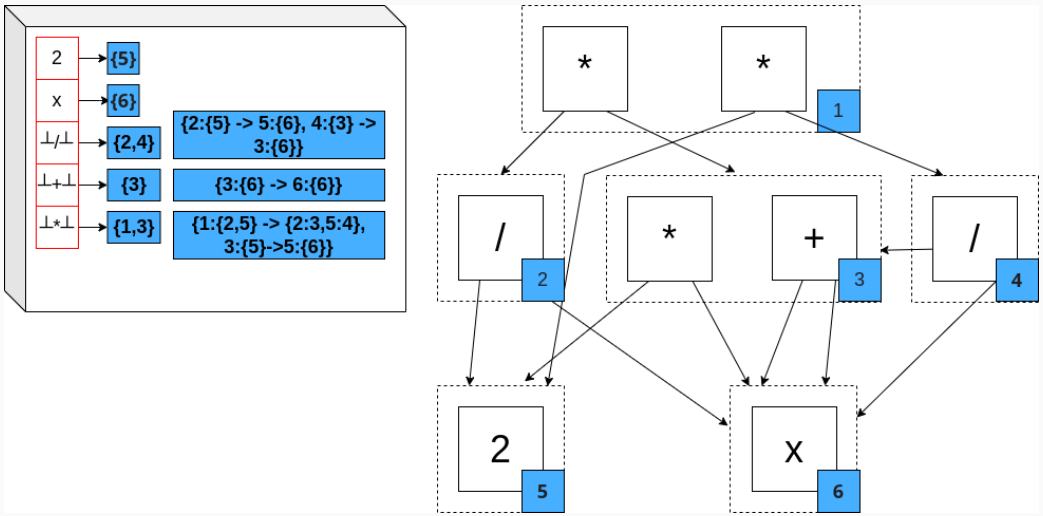
Equality Saturation

Finally, $x \cdot 1 \rightarrow x$:



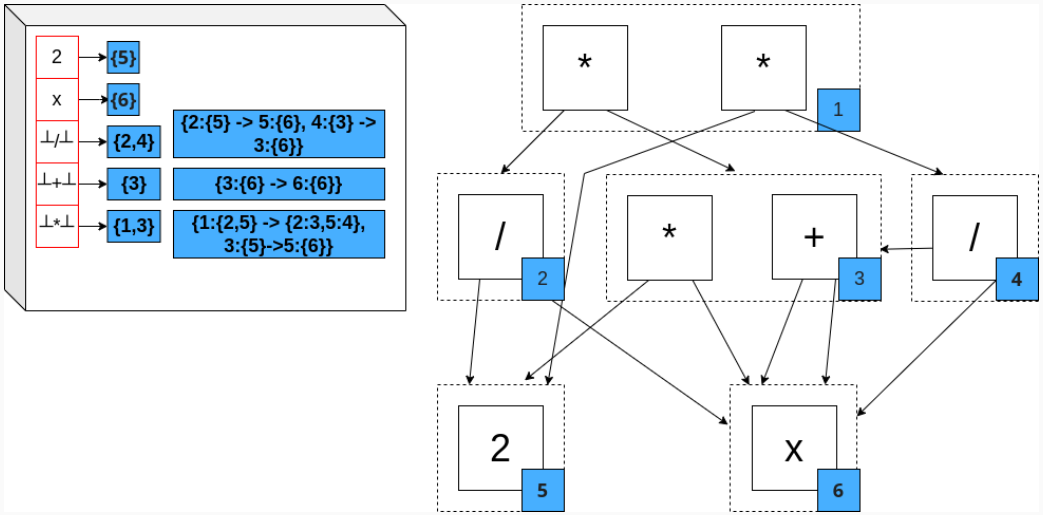
Matching rules

The pattern matching procedure starts with the creation of a database of patterns. This database is composed of a hash table where the keys are each node with their children replaced by a unit value, and the values are a sequence of Tries linking to the e-class of id of each element.



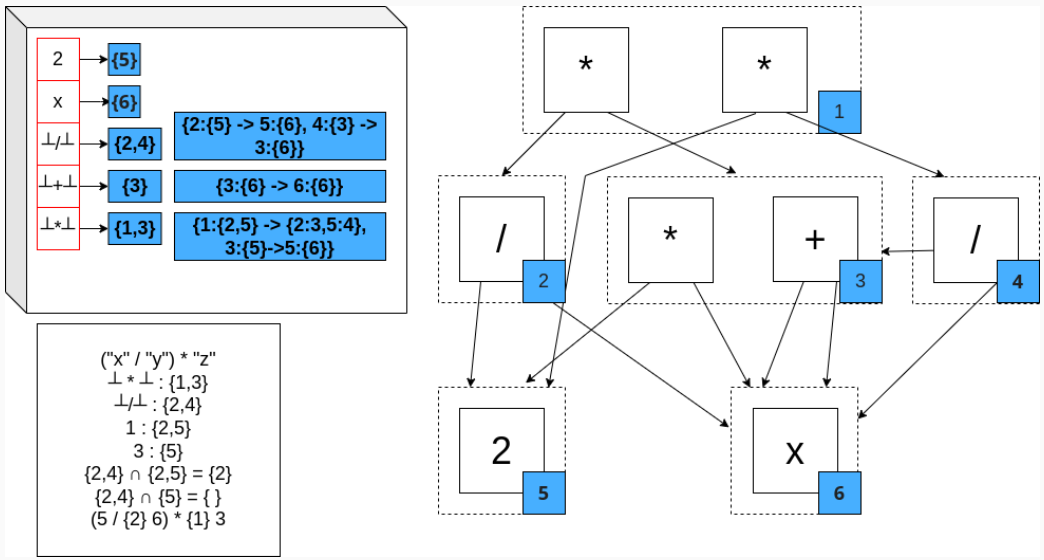
Matching rules

For example, the entry for $*$ contains two e-classes: 1, 3. The first child of e-class 1 can be either 2 or 5 depending of what e-node is chosen. This follows up until for every child of those e-nodes.



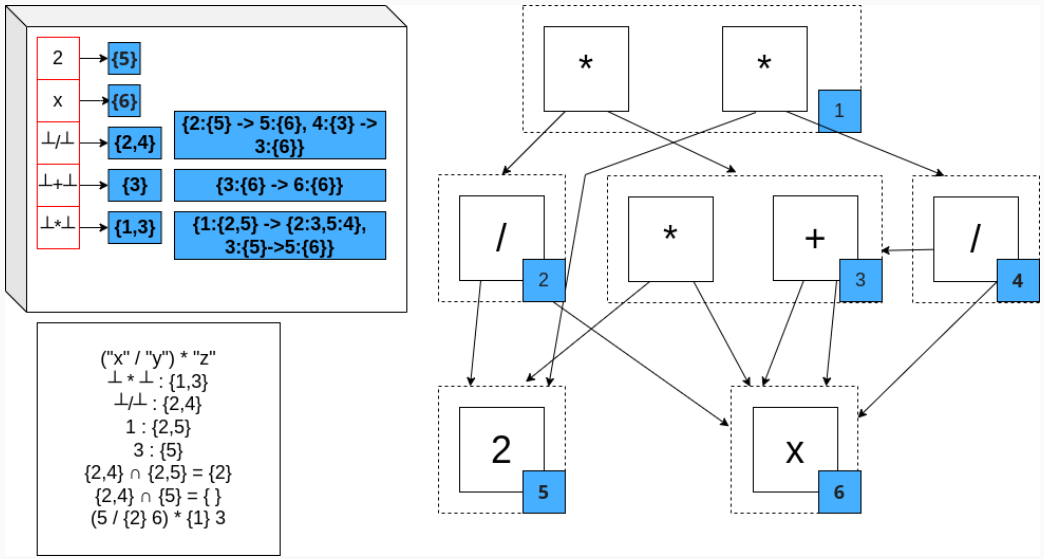
Matching rules

If we want to match the rule `("x" / "y") * "z"`, we first retrieve the patterns `_ * _` from the hash table, finding that it is found at e-classes 1 and 3.



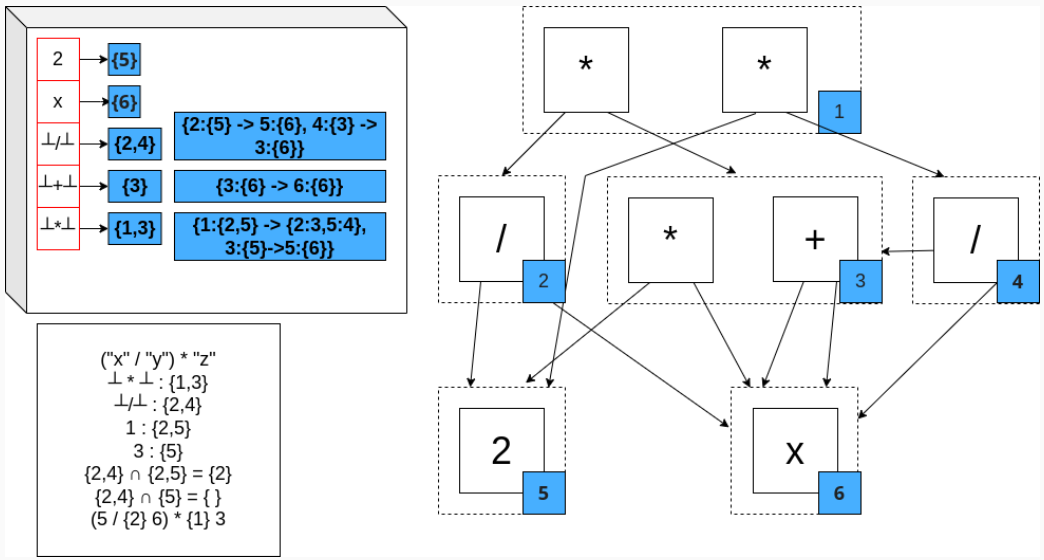
Matching rules

Next, we retrieve the left child of these e-classes, generating the set `{2, 5}`.



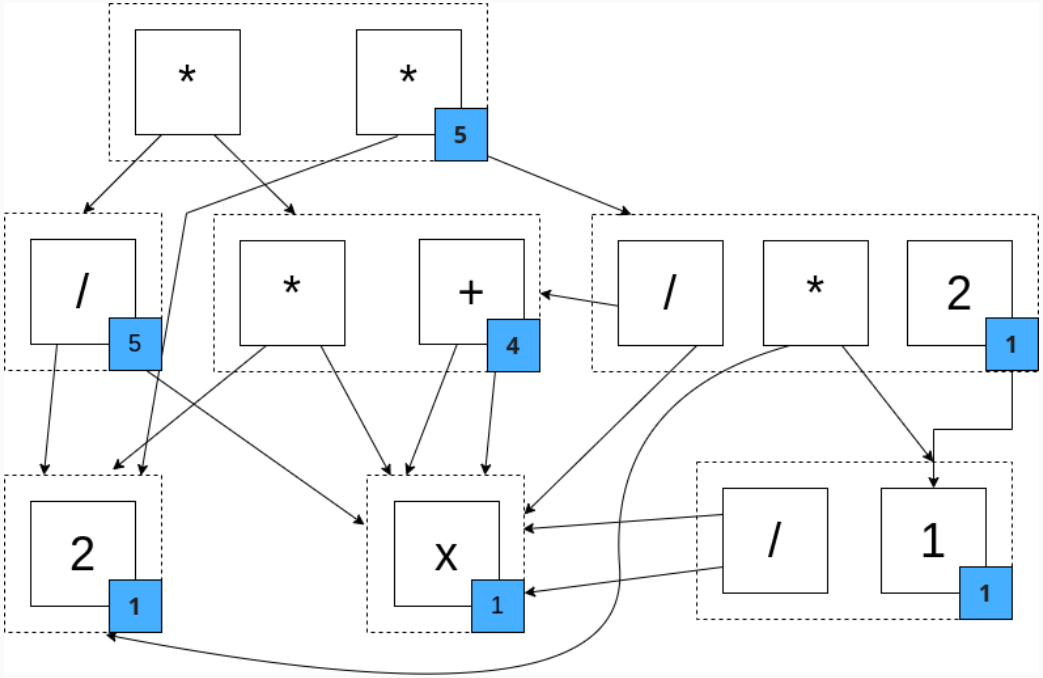
Matching rules

Finally, we retrieve the entry `_ / _` from the hash table, finding the e-classes $\{2,4\}$. Intersecting both sets $\{2,5\} \cap \{2,4\} = \{2\}$ we find the single match.



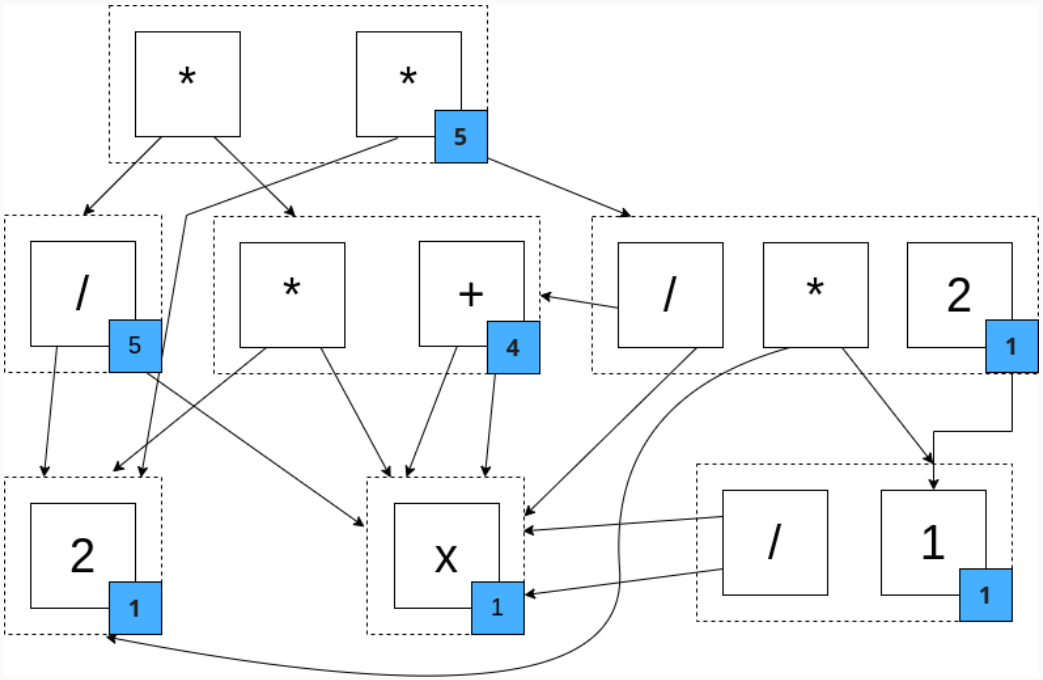
Extracting the Optimal Expression

After reaching saturation, we can extract the optimal expressions by following a greedy heuristic or any graph traversal algorithm we find suitable.



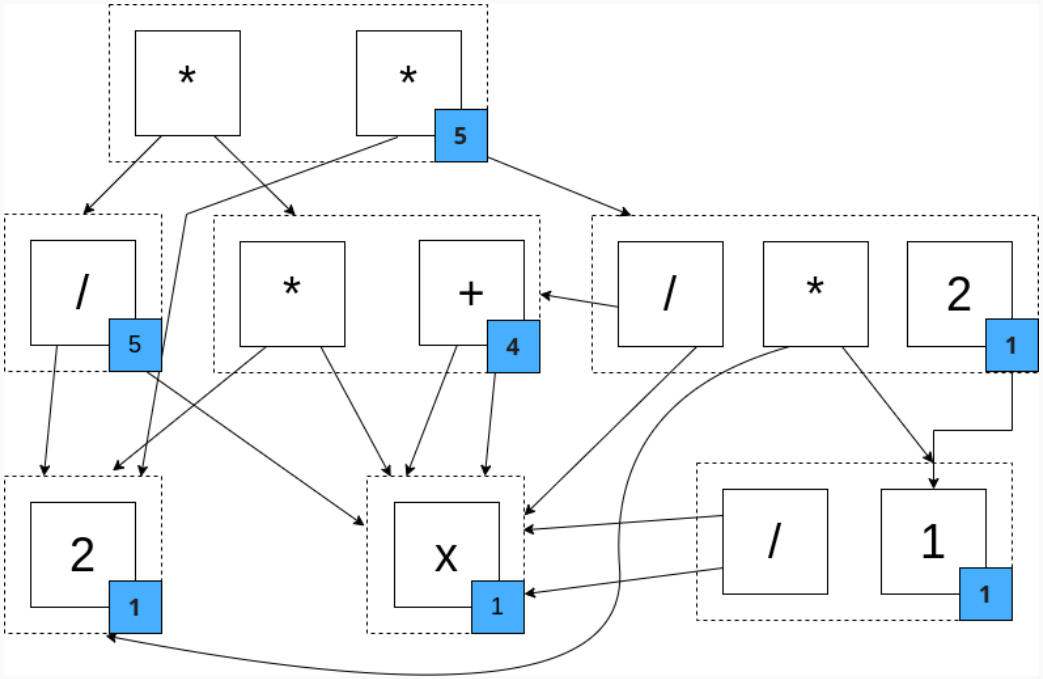
Extracting the Optimal Expression

For example, if we say that $+$ has cost 2 and $*$, $/$ have cost 3, and terminals have cost 1. By greedily following the post-order traversal, we would assign a partial result for each e-class as the minimum result from all e-nodes.



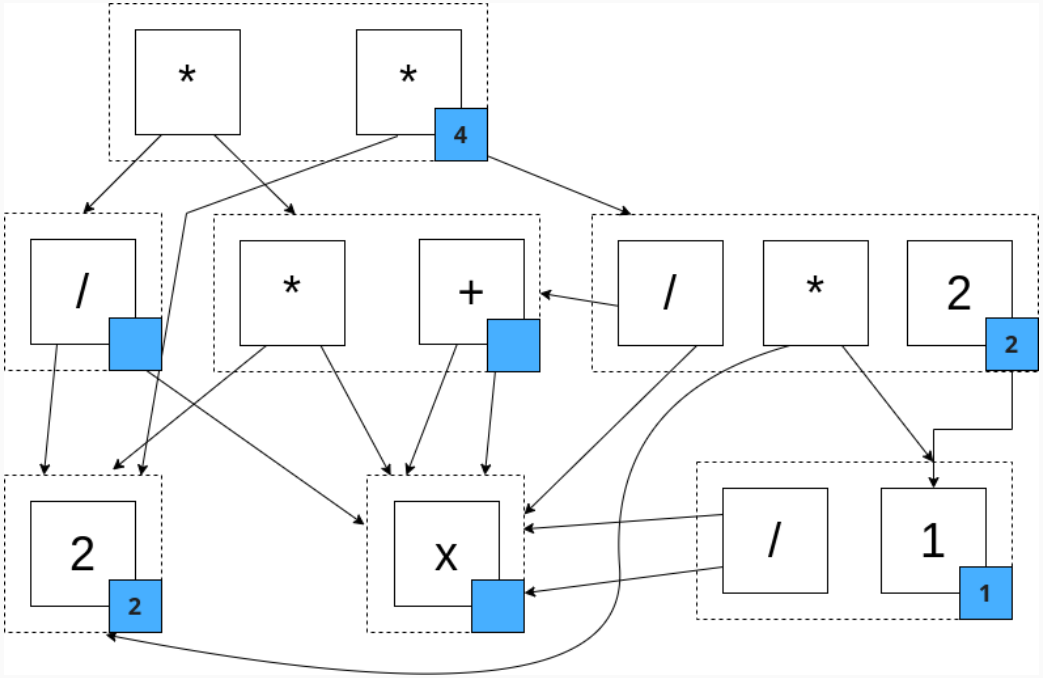
Extracting the Optimal Expression

In our example, we would find $2 * 2$ as the best expression.



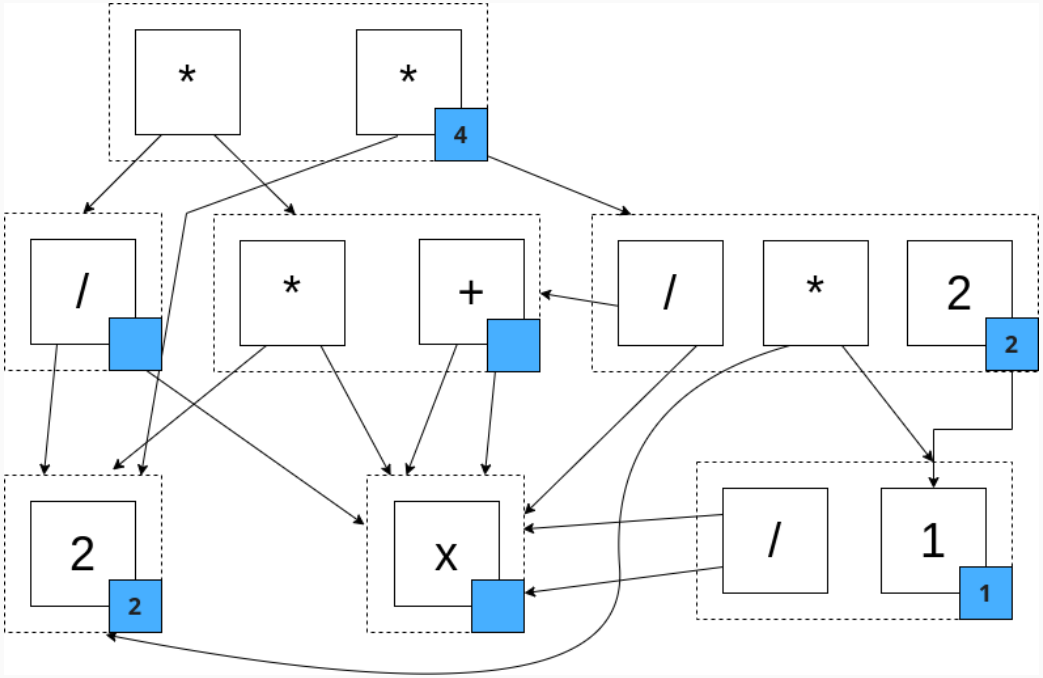
Constant Folding

Other interesting features from *egg* is the constant folding, which allows us to store facts about each e-class. In our example, we can store the constant that this particular e-class can be evaluated to, if any.



Constant Folding

In this situation, we could simply add the e-node 4 into the root e-class, finding an even smaller expression.



Nonterminating rules

- “General characteristic of equality saturation: either a successful rewrite sequence is found relatively quickly, or, computational costs explode” [4]

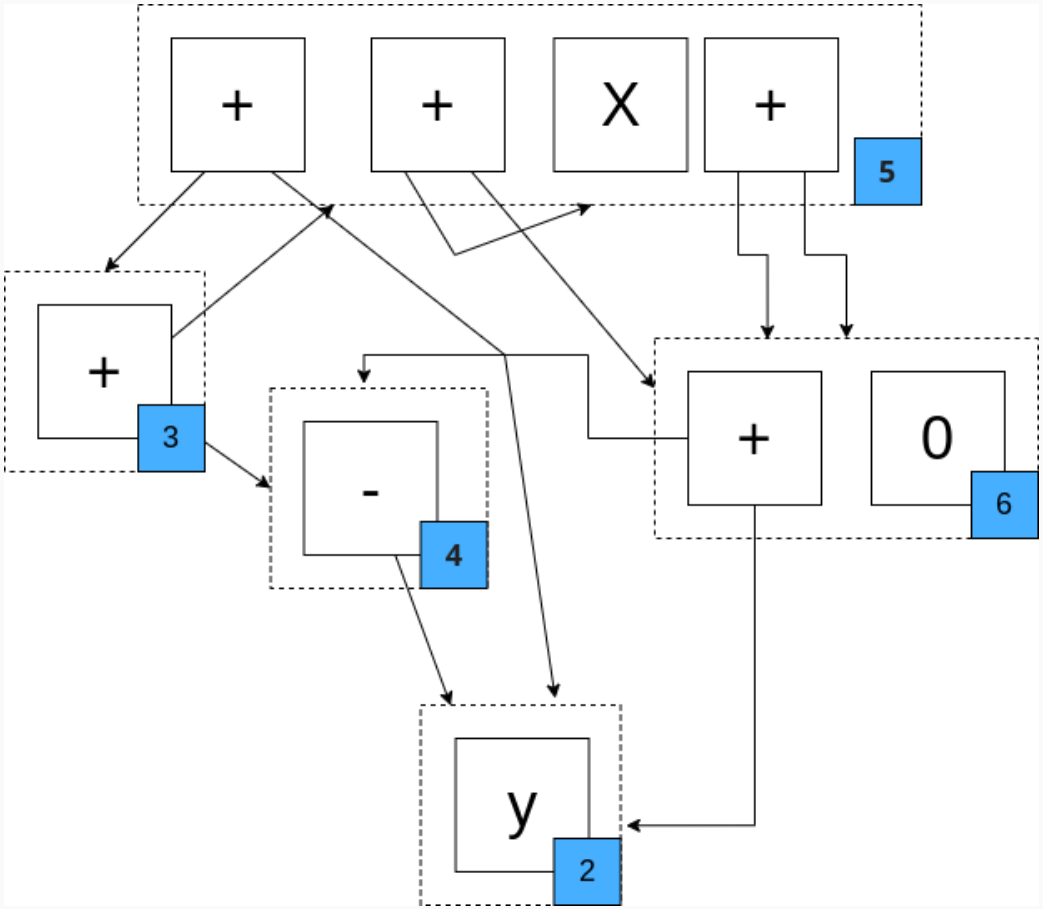
[4] Thomas Kœhler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. Proc. ACM Program. Lang. 8, POPL, Article 58 (January 2024), 32 pages. https://doi.org/10.1145/3632900

Nonterminating rules

Take as an example the rules $(\alpha + \beta) + \gamma \Rightarrow \alpha + (\beta + \gamma)$, $\alpha + 0 \Rightarrow \alpha$, $(-\alpha) + \alpha \Rightarrow 0$. If we have the expression $(x + (-y)) + y$, it will generate:

$$\begin{aligned}(x + (-y)) + y &\Rightarrow x + ((-y) + y) \\ &\Rightarrow (x + (-y)) + y) + (-y + y) \\ &\Rightarrow \ldots\end{aligned}$$

Nonterminating rules



Additional resources (clickable links)

- egraphs-good website
- egg
- hegg (Haskell)
- Metatheory.jl (Julia)
- egg presentation

Application in Symbolic Regression

Overparameterized models

Symbolic Regression algorithms are prone to overparametrization[5]:

$$f(\mathbf{x}, \theta) = \theta_1 \exp(\theta_2 x_1 + \theta_3)$$

- The parameters can assume different values for the same data
- Numerical issues and slow convergence of optimization
- Larger search space for memetic approaches
- Interpretation is hindered

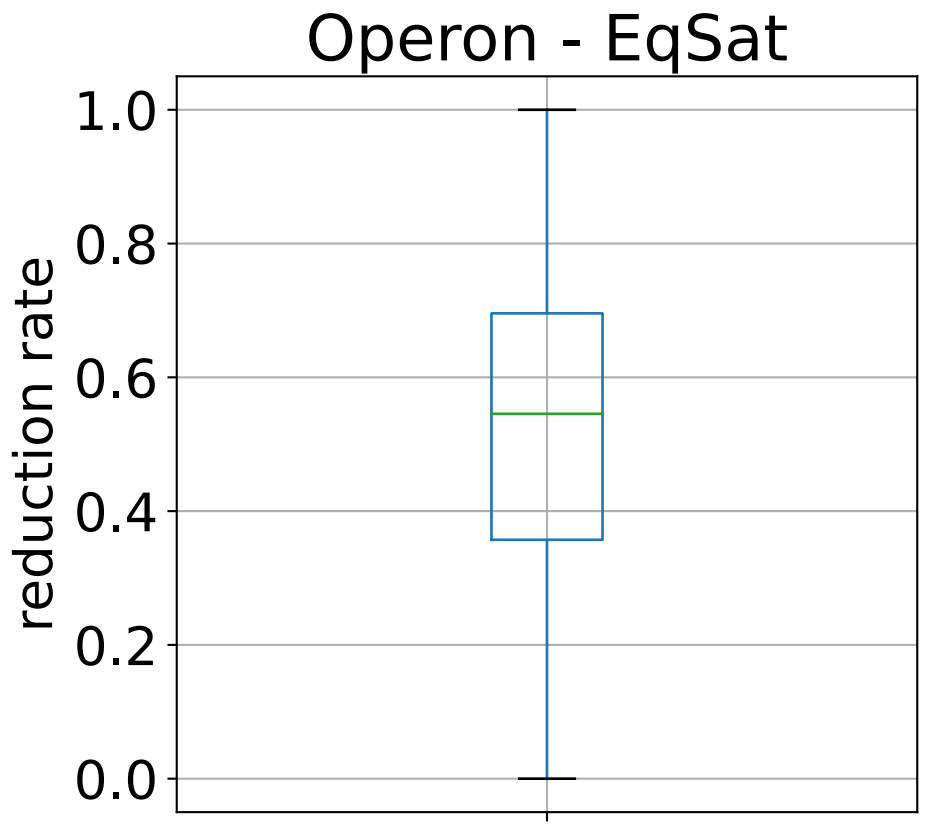
[5] de Franca, Fabricio Olivetti, and Gabriel Kronberger. "Reducing Overparameterization of Symbolic Regression Models with Equality Saturation." Proceedings of the Genetic and Evolutionary Computation Conference. 2023.

Research Questions

- Is overparametrization really a problem?
- Can EqSat help to reduce overparametrization in SR?
  - How well does it compare with simpler alternatives? (e.g., Sympy)
- Does EqSat always reduce to the optimal number of parameters?
- How fast is EqSat?

Is this a real problem?

- Run MOO version of Operon once for every Feynman dataset
- Stored the Pareto front with a total of 183 491 models



How much can EqSat help?

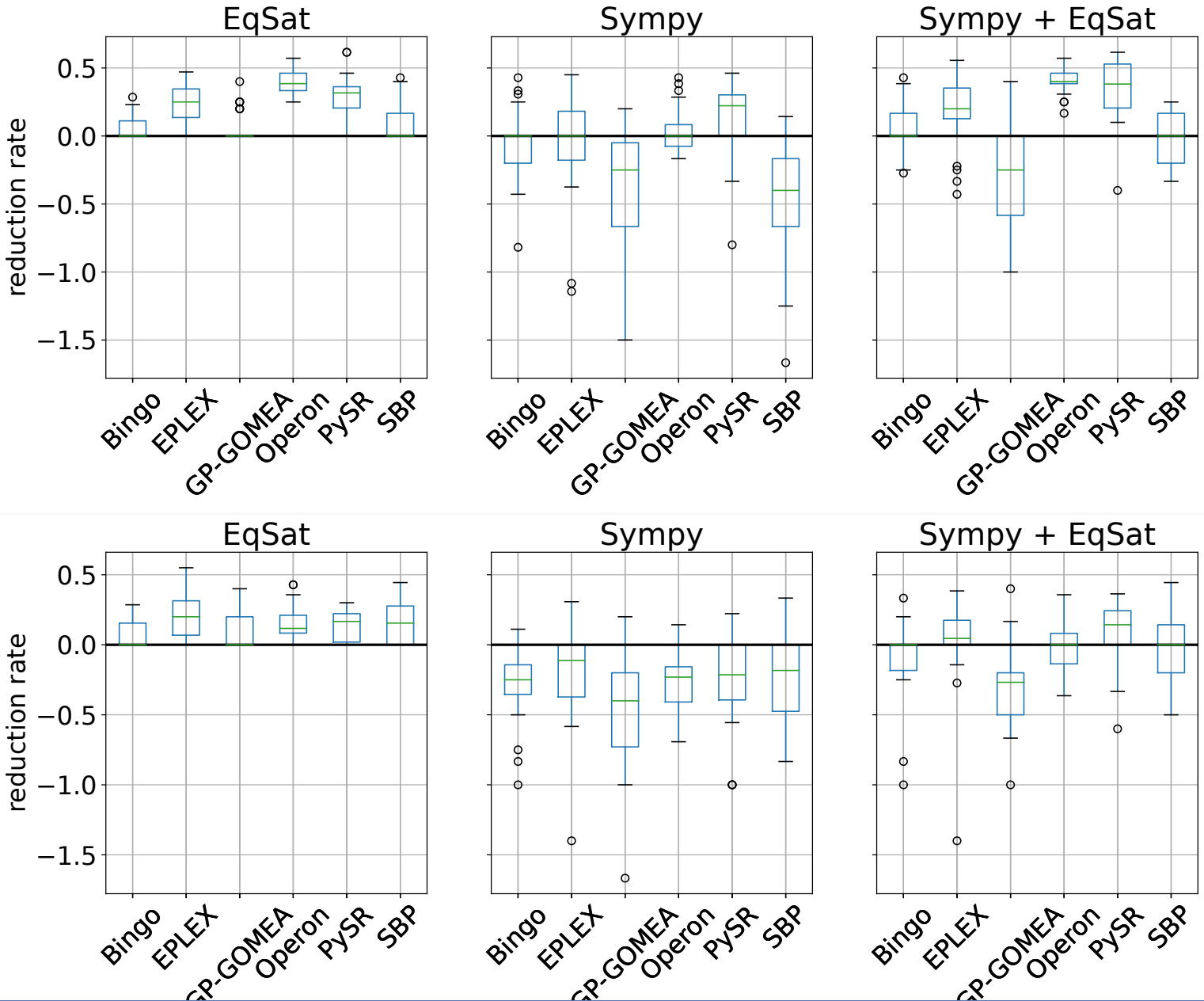
- 30 independent runs of a set of SR algorithms to:

$$f_1(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}} \tag{1}$$

$$f_2(x, y) = \frac{e^{-(x-1)^2}}{1.2 + (y - 2.5)^2} \tag{2}$$

- Simplify with:
  - EqSat
  - Sympy
  - Sympy + EqSat
- SR algorithms:
  - Bingo
  - EPLEX
  - GP-GOMEA
  - Operon
  - PySR
  - SBP

How much can EqSat help?



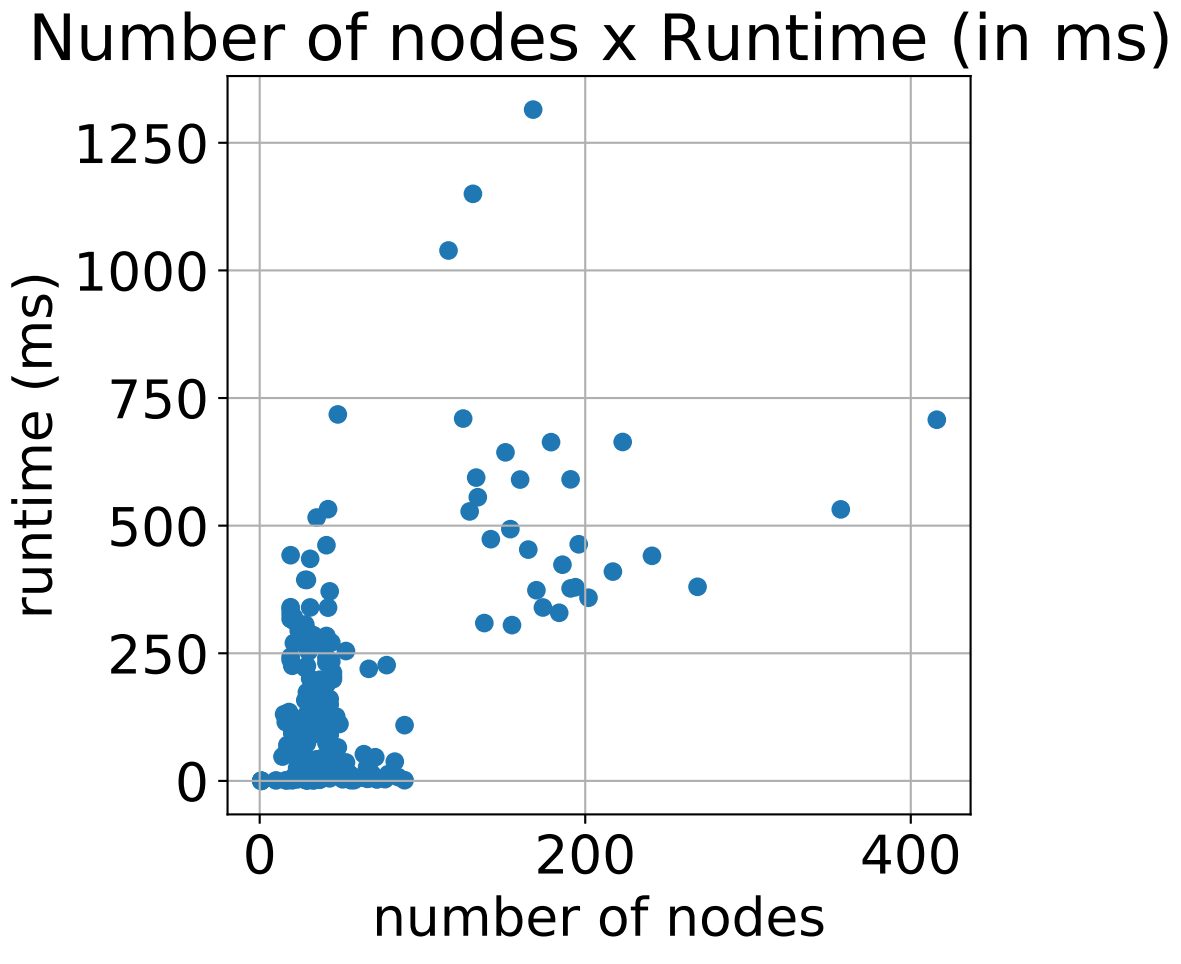
Is it optimal?

% simplified expressions with # parameters equal to rank (left) or with at most one extra (right).

Algorithm	Pagie-1	Kotanchek
Bingo	27%	22%
EPLEX	28%	**18%**
GP-GOMEA	30%	**76%**
Operon	66%	74%
PySR	36%	34%
SBP	42%	60%

Algorithm	Pagie-1	Kotanchek
Bingo	**33%**	66%
EPLEX	45%	37%
GP-GOMEA	**100%**	**100%**
Operon	**100%**	94%
PySR	52%	71%
SBP	60%	**100%**

Is it fast?


Number of nodes x Runtime (in ms)
runtime (ms)
1250
1000
750
500
250
0
0
200
400
number of nodes

Discussions

- Overparametrization is widespread in SymReg
- EqSat can consistently reduce the number of parameters
- Sympy sometimes makes it worse
- Runtime is low most of the time and it can be improved

Next lecture

- Confidence Intervals with Profile Likelihood

Acknowledgments