E-Graphs and Equality Saturation (in Haskell)

Rodrigo Mesquita

March 8, 2023

E-graphs and eq-sat are cool

Published applications of equality saturation and e-graphs

- Automatic vectorization of digital signal processing code
- Tensor graph superoptimization
- Algebraic metaprogramming and symbolic computation
- and more...

E-graphs and eq-sat are cool

And perhaps in the near future...

- A symbolic mathematics library in Haskell
- Pattern-match coverage checking in GHC
- <Insert your idea here>

What are e-graphs?

Definition (E-graphs)

- An e-graph is a data structure that
  - compactly represents *equivalence classes* of expressions
  - while maintaining a key invariant: the equivalence relation is closed under *congruence*[a]
- Concretely, an e-graph is a set of equivalence classes.
  - Each e-class is a set of equivalent e-nodes
  - An e-node represents a expression from a given language (e.g. $x * 1$)
  - An e-node is a function symbol paired with a list of children e-classes $f(c_1, c_2, \ldots)$

[a] Intuitively, $a \equiv b \Rightarrow f(a) \equiv f(b)$

What are e-graphs?

Definition (E-graphs)

- An e-graph is a data structure that
  - compactly represents *equivalence classes* of expressions
  - while maintaining a key invariant: the equivalence relation is closed under *congruence*[a]
- Concretely, an e-graph is a set of equivalence classes.
  - Each e-class is a set of equivalent e-nodes
  - An e-node represents a expression from a given language (e.g. $x * 1$)
  - An e-node is a function symbol paired with a list of children e-classes $f(c_1, c_2, \ldots)$

[a] Intuitively, $a \equiv b \Rightarrow f(a) \equiv f(b)$

What are e-graphs?

Definition (E-graphs)

- An e-graph is a data structure that
  - compactly represents *equivalence classes* of expressions
  - while maintaining a key invariant: the equivalence relation is closed under *congruence*[a]
- Concretely, an e-graph is a set of equivalence classes.
  - Each e-class is a set of equivalent e-nodes
  - An e-node represents a expression from a given language (e.g. $x * 1$)
  - An e-node is a function symbol paired with a list of children e-classes $f(c_1, c_2, \ldots)$

[a] Intuitively, $a \equiv b \Rightarrow f(a) \equiv f(b)$

What are e-graphs?

Definition (E-graphs)

- An e-graph is a data structure that
  - compactly represents *equivalence classes* of expressions
  - while maintaining a key invariant: the equivalence relation is closed under *congruence*[a]
- Concretely, an e-graph is a set of equivalence classes.
  - Each e-class is a set of equivalent e-nodes
  - An e-node represents a expression from a given language (e.g. $x * 1$)
  - An e-node is a function symbol paired with a list of children e-classes $f(c_1, c_2, \ldots)$

---

[a] Intuitively, $a \equiv b \Rightarrow f(a) \equiv f(b)$

What are e-graphs?

Definition (E-graphs)

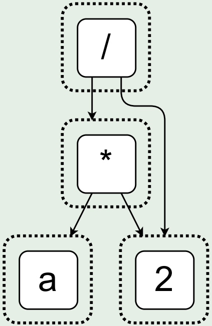
- An e-graph is a data structure that
  - compactly represents *equivalence classes* of expressions
  - while maintaining a key invariant: the equivalence relation is closed under *congruence*[a]
- Concretely, an e-graph is a set of equivalence classes.
  - Each e-class is a set of equivalent e-nodes
  - An e-node represents a expression from a given language (e.g. $x * 1$)
  - An e-node is a function symbol paired with a list of children e-classes $f(c_1, c_2, \ldots)$

[a] Intuitively, $a \equiv b \Rightarrow f(a) \equiv f(b)$

E-graphs represent congruence relations over expressions

Example (E-graph)

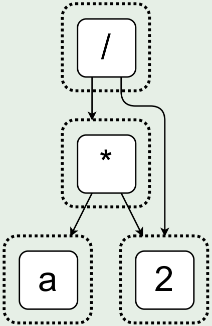
- This e-graph represents $(a * 2)/2$
- In fact, it represents a, 2 and $a * 2$ too



E-graphs represent congruence relations over expressions

Example (E-graph)

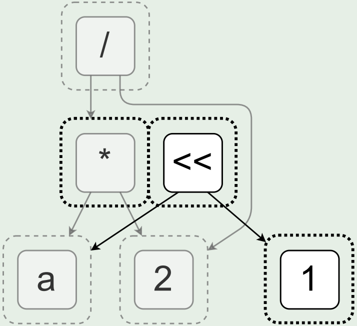
- This e-graph represents $(a * 2)/2$
- In fact, it represents a, 2 and $a * 2$ too



E-graphs represent congruence relations over expressions

Example (E-graph)

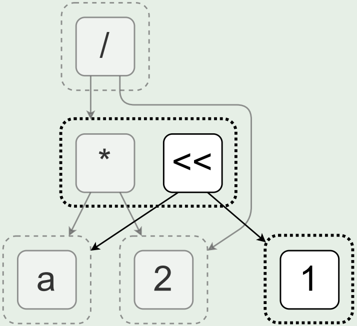
- Now we **represent** the expression $a \ll 1$ in a new e-class
- And **merge** that e-class with the e-class representing $a * 2$
  - We merge these two classes because they represent equivalent expressions
  - By congruence, $(a * 2)/2$ is now seen as equivalent to $(a \ll 1)/2$



E-graphs represent congruence relations over expressions

Example (E-graph)

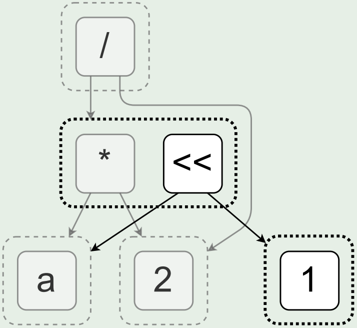
- Now we **represent** the expression $a \ll 1$ in a new e-class
- And **merge** that e-class with the e-class representing $a * 2$
  - We merge these two classes because they represent equivalent expressions
  - By congruence, $(a * 2)/2$ is now seen as equivalent to $(a \ll 1)/2$



E-graphs represent congruence relations over expressions

Example (E-graph)

- Now we **represent** the expression $a \ll 1$ in a new e-class
- And **merge** that e-class with the e-class representing $a * 2$
  - We merge these two classes because they represent equivalent expressions
  - By congruence, $(a * 2)/2$ is now seen as equivalent to $(a \ll 1)/2$



What about equality saturation?

Definition (Equality Saturation)

- Eq-sat is a technique that leverages e-graphs to implement term rewriting systems
- In short, we
  - Represent an expression in the e-graph
  - Repeatedly apply pattern-based rewrites until saturation (rewrite rules like $x * 1 \to x$ where x matches any expression)
  - Extract the best expression equivalent to the input

What about equality saturation?

Definition (Equality Saturation)

- Eq-sat is a technique that leverages e-graphs to implement term rewriting systems
- In short, we
  - Represent an expression in the e-graph
  - Repeatedly apply pattern-based rewrites until saturation (rewrite rules like $x * 1 \to x$ where x matches any expression)
  - Extract the best expression equivalent to the input

What about equality saturation?

Definition (Equality Saturation)

- Eq-sat is a technique that leverages e-graphs to implement term rewriting systems
- In short, we
  - Represent an expression in the e-graph
  - Repeatedly apply pattern-based rewrites until saturation (rewrite rules like $x * 1 \to x$ where x matches any expression)
  - Extract the best expression equivalent to the input

What about equality saturation?

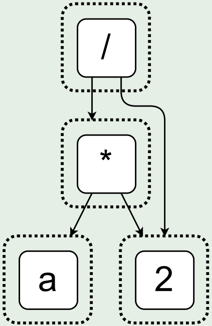
Definition (Equality Saturation)

- Eq-sat is a technique that leverages e-graphs to implement term rewriting systems
- In short, we
  - Represent an expression in the e-graph
  - Repeatedly apply pattern-based rewrites until saturation (rewrite rules like $x * 1 \to x$ where x matches any expression)
  - Extract the best expression equivalent to the input

Example (Rewriting $(a * 2)/2$)

Rewrite rules:

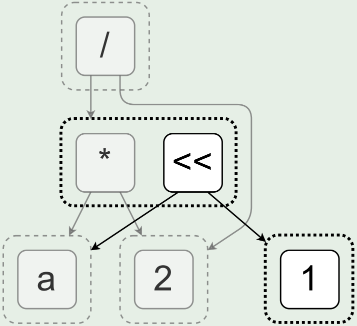
- $a * 2 = a \ll 1$
- $(a * b)/c = a * (b/c)$
- $x/x = 1$
- $x * 1 = x$



Example (Rewriting $(a * 2)/2$)

Rewrite rules:

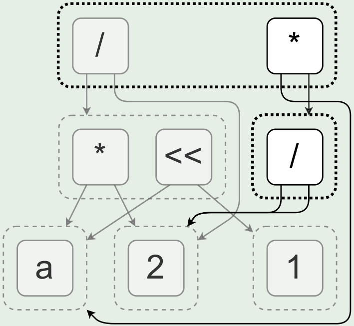
- $a * 2 = a \ll 1$
- $(a * b)/c = a * (b/c)$
- $x/x = 1$
- $x * 1 = x$



Example (Rewriting $(a * 2)/2$)

Rewrite rules:

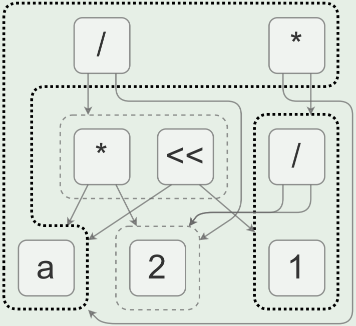
- $a * 2 = a \ll 1$
- $(a * b)/c = a * (b/c)$
- $x/x = 1$
- $x * 1 = x$



Example (Rewriting $(a * 2)/2$)

Rewrite rules:

- $a * 2 = a \ll 1$
- $(a * b)/c = a * (b/c)$
- $x/x = 1$
- $x * 1 = x$



Fast and extensible equality saturation[1]

- Invariant restoration technique called *rebuilding*
- Mechanism called *e-class analysis* to integrate domain specific analysis into the e-graph

[1]egg: Fast and Extensible Equality Saturation (POPL 2021)

Rebuilding

The key idea: defer e-graph invariant restoration to the call of *rebuild*

- Under certain workloads (automated theorem provers such as Z3), e-graph invariants are restored after every operation
- For equality saturation, we can greatly benefit from deferred invariant maintenance:
  - Search for all pattern matches
  - Apply all rewrites that matched[2]
  - Restore invariants[3]

[2] We can't search patterns while applying rewrites – the invariants aren't maintained w/out **rebuild**

[3] Potentially saves work because we deduplicate the *worklist*

Rebuilding

The key idea: defer e-graph invariant restoration to the call of *rebuild*

- Under certain workloads (automated theorem provers such as Z3), e-graph invariants are restored after every operation
- For equality saturation, we can greatly benefit from deferred invariant maintenance:
    - Search for all pattern matches
    - Apply all rewrites that matched[2]
    - Restore invariants[3]

[2] We can't search patterns while applying rewrites – the invariants aren't maintained w/out **rebuild**
[3] Potentially saves work because we deduplicate the *worklist*

E-class Analysis

The key idea: attach data to each e-class and use it to modify the e-graph

- Purely syntactic rewrites are insufficient in certain applications, in which we need domain knowledge (e.g. constant folding)
- Previously, this knowledge was added by modifying the e-graph with ad-hoc passes
- E-class analysis allows the expression of analysis over the e-graph
    - Each e-class has data from a semilattice
    - Merging e-classes merges the data
    - We can use the data to modify the e-graph
    - (See interface of Analysis)

E-class Analysis

The key idea: attach data to each e-class and use it to modify the e-graph

- Purely syntactic rewrites are insufficient in certain applications, in which we need domain knowledge (e.g. constant folding)
- Previously, this knowledge was added by modifying the e-graph with ad-hoc passes
- E-class analysis allows the expression of analysis over the e-graph
  - Each e-class has data from a semilattice
  - Merging e-classes merges the data
  - We can use the data to modify the e-graph
  - (See interface of Analysis)

E-class Analysis

The key idea: attach data to each e-class and use it to modify the e-graph

- Purely syntactic rewrites are insufficient in certain applications, in which we need domain knowledge (e.g. constant folding)
- Previously, this knowledge was added by modifying the e-graph with ad-hoc passes
- E-class analysis allows the expression of analysis over the e-graph
  - Each e-class has data from a semilattice
  - Merging e-classes merges the data
  - We can use the data to modify the e-graph
  - (See interface of Analysis)

E-graphs in hegg (Demo)

Remark

Follow through with the documentation: Data.Equality.Graph

- Why are expressions in their functorial form?

```haskell
data SymExpr a
    = Const Double
    | Symbol String
    | a :+: a
    | a :*: a
    | a :/: a
    deriving (Functor, Foldable, Traversable)
```

References

- egg: Fast and Extensible Equality Saturation
- Relational E-matching