Generic E-Graph in Lean4

Junyong Heo

This report summarises the work done for the Large Research Project Computer Science (XM_0130). Sections 1-3 touch shortly on the goals of the project and the design choices made in light of them. Section 4 is a quick run-through of e-graph implementation. Sections 5-6 show examples on how to run the code. Sections 7-8 discuss the limitations of the system and future improvements. The appendix holds a list of useful functions for the user.

Contents

1. Introduction

1.1. **Problem Statement.**

When transforming a given input through a series of rewrites or optimisations, we often run into the problem that the order in which the rewrites are performed has an effect on the final value. This is often referred to as the *phase ordering problem*, and stems from the fact that rewrites are inherently destructive; take the example $(a \times 2) / 2$. Multiplying by a factor of 2 can more efficiently be done through bit shifts, and thus we apply $(a \times 2) / 2 \to (a \ll 1) / 2)$. While the rewrite leads us to a more efficient operation, it fails to see the global optimum of $(a \times 2) / 2 \to a \times (2 / 2) \to a$. Equality saturation uses an e-graph to maintain classes of equivalent terms, allowing for non-destructive rewrites to sidestep this issue [1].

While there currently exists an implementation of equality saturation and e-graphs for Lean 4 (hereafter referred to as Lean) internal terms in Lean, there is not yet a generic equality saturation framework adaptable to different domains and languages. This project aims to bridge that gap by developing a generic e-graph library in Lean, as well as a framework for equality saturation over a user-defined language, with support for rewrite rules and extraction.

As it is, the library is not fully formalised. The project serves as an exploratory introduction into the subject and to prepare a foundation for a long term goal of enabling formally verified optimisation and equational reasoning for domain specific languages.

1.2. **Contributions.**

Specifically, this project makes the following contributions.

- A Generic E-Graph framework in Lean parameterised over a user-defined language
  - E-graph core (E-Node, E-Class, and E-Graph operations)
  - Pattern matching and (conditional) rewriting
  - Extensible execution framework and equality saturation
  - E-class analysis and cost extraction
- Documentation and examples over a few domains

2. Background

A very short background for completeness' purposes.

2.1. **Lean 4.**

Lean is an interactive theorem prover and functional programming language based on dependent type theory and inductive constructions. In addition to mathematics, Lean can be used to prove properties about programs. It's powerful metaprogramming capabilities make it useful for defining and working with domain specific languages.

2.2. **E-Graph.**

E-Graphs are a representation of terms and congruence relations of these terms [1]. More precisely, an e-graph uses a union-find data structure to maintain equivalence relations of e-classes, which themselves hold a set of equivalent e-nodes. Therefore, rather than storing a single expression, e-graphs make it possible to compactly store multiple equivalent expressions, making it suitable for equality saturation and rewriting based optimisations.

Willsey et al. define an e-graph as a tuple (U, M, H) where:

- U: A union-find data structure over e-class ids.
- M: An e-class map that maps e-class ids to e-classes
- H: A hashcons map from e-nodes to e-class ids.

Details related to implementation will be elaborated in a further section.

3. Design

In this section we detail some overarching design choices made. Specific design decisions of note may be elaborated more in the appropriate section.

3.1. **Simple vs Dependent Typing.**

In this project, we elected to implement the e-graph in a simply typed form over a dependently typed one. The main motivation is that the e-graph was planned with functionality in mind, and while dependent typing can help preserve invariants throughout, simple typing makes it easier to reason about the logic without getting drowned in implementation details. Furthermore, egg-style deferred rebuilding often results in invariants being temporarily broken, such as non-canonical references and a stale hashcons. Therefore, we elect to separate the proofs from the core functionality, while still allowing proving invariants in the form of separate theorems later. A core benefit is therefore more efficient execution through ease of merging e-classes, updating the union find, and allowing to stay closer to egg's algorithms.

3.2. **Stateful Computation.**

Instead of a fully functional structure, we thread the e-graph state through computations. This choice is again mostly motivated by the focus on practicality and execution.

3.3. **Genericity.**

To allow the e-graph to work for any user defined language, we generalise the structure and put the burden of proof and implementation on the user. No assumptions and simplification on the language are made, and as such the user is free to introduce rules and constructs as needed. Additionally, the e-graph can only represent syntactic equivalence and rewrite rules, and the proof of rewrite rules must be provided by the user.

To help on the semantic rewriting, analysis can be extended within reason. However, we also don't enforce invariants and thus nothing stops the user from defining an ill-formed analysis, as I learned from my own testing.

4. Implementation

In this section, we discuss implementation details. The words *list* and *array* are used interchangeably to refer to a generic collection. The words *set* and *map* are used in the normal sense; *set* is used where uniqueness of the elements is important and *map* for collections where one value is indexed by the other.

4.1. **Language.**

While the e-graph is parameterised over a generic language α, we put constraints of DecidableEq, Hashable, and Repr. Hashable is used in the hashcons, as well as throughout the implementation in the form of a hashmap. While boolean equality suffices for the implementation we currently have, decidable equality was chosen to open up the possibility of verifying details about the data structure later.

Furthermore, every structure derives Repr for easier debugging and visualisation. Lean should take care of most of these automatically, but it may be required to define them yourself.

4.2. **E-Graph Core.**

The core e-graph data structure consists of the e-node, e-class, and e-graph structures, as well as the *(U,M,H)* triple mentioned in section Section 2.2. *EClassId* is used as a synonym for *Nat* for clarity.

4.2.1. *E-Node.*

An e-node consists of:

- *head* of type α.
- List *args* of type EClassId.

The head represents a construct of the language, whether it be a primitive or an operator. For example, the term $(2 + 3)$ of type MathLang will have '+' : MathLang as the head and the ids of the canonical e-classes that hold the operands 2 and 3 in args.

4.2.2. *E-Class.*

At its core, an e-class is nothing more than a collection of syntactically equivalent e-nodes. In addition, we maintain a list of parents, to better preserve the invariants required by the e-graph, improve efficiency through hashconsing, and efficient bookkeeping with merges.

To allow for analysis, we also provide a *data* field of type D that is extendable by the user.

4.2.3. *E-Graph.*

The e-graph consists of:

- Union-find data structure over e-class ids.
- Map from e-class ids to e-classes
- Hashcons of e-nodes to e-class ids

as well as two additions,

- List of "dirty" e-classes
- Map from operators to e-class ids

The union-find represents equivalences between e-classes and provides a suitable framework to for querying and union. The e-class map bridges the e-class ids in the union-find and the actual e-classes.

The "dirty" list keeps track of classes that need rebuilding, allowing us to mark classes for the deferred batch rebuilding. The operator map, a minor optimisation to reduce the search space during pattern matching, will be discussed more in Section 4.3.4.

4.2.4. *Operations.*

The two main operations exposed to the user are the *push* and *union* operations. The *push* operation introduces a new e-node to the e-graph, and *union* marks an equivalence relation between two nodes, modifying the e-graph appropriately.

4.3. **Rewriting.**

Rewriting consists of two main parts, pattern matching and rewriting.

4.3.1. *Pattern Matching.*

Pattern matching consists of taking a pattern with variables and finding complete terms that fit the pattern. For example, if we try match the pattern *?x + 0 → ?x*, against terms in the graph, we might find *foo(a,b) + 0* fits the pattern.

Intuitively, we can view it in this way: if we think of the terms as a syntax tree, we first find a node whose operator matches that of the pattern we need. In our example above, we find an operator node with value '+'. Then we look at its children, of which one is a "hole" and the other is a constant 0. If we see that the right child matches 0, then we can store the subtree of the left child as a match for the "hole". This is illustrated in the not very beautiful illustration below.

FIGURE 1. Syntax tree visualisation for e-matching

After a rule finishes matching, we end up with a dictionary that shows all terms in the e-graph that fit the hole. A dictionary for a pattern with multiple holes would therefore look like:

```
Dict := { (?x, ?y) => [(foo,bar), (23, 50)] }
```

4.3.2. *Rewriting.*

Rewriting takes the dictionary created above to create syntactically equivalent expressions. Given a rewrite rule that says *?x + ?y => ?y + ?x* , we take a pair of terms from the dictionary above and rearrange them according to the rewrite. Therefore, *foo + bar* is rewritten into *bar + foo* , and the appropriate node is added to the graph.

It's important to remember that as e-graphs work across a **class** of equivalent terms, both the dictionary and the rewrites work on e-class ids and not nodes. Therefore, a more appropriate representation would be *(EClass 5) + (EClass 7) => (EClass 7) + (EClass 5)* .

4.3.3. *Conditional Rewriting.*

As a bonus, we implemented an extendable conditional rewriting framework. The user can define and pass function of type *Dict → EGraphM Lang Analysis Bool* which takes the dictionary created in the pattern matching step that evaluates the entries. While this is most useful in conjunction with analysis, for example only rewriting if the e-class represents a constant, the user may pass on any evaluator.

4.3.4. *(Operator Indexed) Pattern Matching.*

A simplification made is the choice of pattern matching algorithm. The e-matching algorithm implemented in this project is a backtracking search across the entire e-graph. However, to somewhat reduce the search space, the e-graph maintains a map of operators to e-class ids. For example, over a calculus-based language, the operator-map (or opmap), keeps track of the ids of all e-classes that contain a differentiation term. When we try and match a differentiation rule, we only need to search the e-classes indexed by the opmap. However, this results in an non-uniform speedup across operators, as the benefits of the technique are most visible in sparsely used operators. In our example, matching an addition rule will likely show a much slower speedup compared to naïve matching than differentiation, as equivalent addition based terms can come from rewrites of many

other operations. We will discuss further improvements to the pattern matching in a later section.

4.3.5. *Equality Saturation.*

An equality saturation function is also provided. Similar to egg, we implement phase separation, dividing each iteration into a search phase and rewriting phase. An obvious benefit is that this provides better parallelisation, but it also reduces the dependence on rewrite rule application order and makes each iteration more predictable.

In addition to egg's Search → Apply → Rebuild, we rebuild the opmap (operator map) at the beginning of every iteration to keep the information in the map up to date, ensuring we don't miss any potential matches or waste effort on classes that have been canonicalised away.

The Equality Saturation (eqSat) function takes 1 mandatory argument, a list of rewrite rules to use. However, we can also stop the function early by providing either an iteration limit (default: limit := 10) or a node limit argument.

4.3.6. *Extraction.*

We implement a simple bottom up extractor taken from Egg's Extraction Gym [2]. The user can pass a custom cost function, so long as it takes $\alpha \to \text{Nat}$.

5. Running the Code

5.1. **Defining Language and Rules.**

To define a language, first we must define the syntax. An example is given for a basic Add/Mul language.

```
inductive AddMul where
| lit   : Nat → AddMul
| var   : String → AddMul
| add   : AddMul
| mul   : AddMul
deriving DecidableEq, Hashable, Repr

instance : ToString AddMul where
toString
| .lit n   => s!"{n}"
| .var s   => s
| .add     => "+"
| .mul     => "*"

instance Analysis : EGraph.Analysis AddMul Unit where
  make    _  _ := ()
  join    _  _ := ()
  modify  eg _ := eg

abbrev EGraphIO := EGraphGenericIO AddMul Unit
```

Syntax can be defined inductively as shown above. While a base analysis instance is defined, for reasons still unknown to me if one is required the above definition is portable.

A further EGraphGenericIO that takes a language and analysis is provided which wraps the EGraph around a state monad and the IO monad. As most runtime functions take an EGraphIO state, we suggest using it.

To facilitate testing, we provide an extensible s-expr style parsing interface.

```
instance : ParseExpr AddMul where
  parse sx :=
  match sx with
  | .atom s =>
    match s.toNat? with
    | some n => some (.lit n, [])
    | none   => some (.var   s, [])
  | .list (head :: args) =>
    match head with
    | .atom "+"    => some (.add,  args)
    | .atom "*"    => some (.mul,  args)
    | _ => none
  | .list [] => none
```

We also find it useful to define functions to lift syntax into the Pattern class. While this does not force the arity of an operator, it does stop ill-formed rules from being parsed.

To help define rules, we have two functions, liftVar and liftTerm. liftTerm is used to shape our rule definitions, while liftVar is used to represent holes in the rules.

```
def liftVar (s : String) : Pattern MathLang := Pattern.PatVar s
def liftTerm (h : MathLang) (args : List (Pattern MathLang)) : Pattern
MathLang := Pattern.PatTerm h (Array.mk args)


def pAdd  (x y : Pattern AddMul ) := liftTerm (.add    ) [x, y]
def pMul  (x y : Pattern AddMul ) := liftTerm (.mul    ) [x, y]
...
def pLit (n : Int               ) := liftTerm (.lit   n) []
def pVar (s : String            ) := liftTerm (.var s  ) []
```

which allow us to define our rules as such

```
r* pAdd (?"a") (?"b") === pAdd (?"b") (?"a"),
r* pPow (?"x") (pConst 0) === (pConst 1) if (isNotZero "x"),
r* pDiff {LHS} === {RHS} ifMultiple [
  (isNotZero "f"), (isNotZero "g")
]
```

By defining a few macros, we are able to define rules by following a format of "r* {LHS} === {RHS}", followed by either "if {single condition}" or "ifMultiple { [list of conditions] }". Conditional functions may be defined as previously mentioned, with an example format

```
def isConst (var : String) : Dict Lang → EGraphM Lang Analysis Bool :=
  λ (dict) => (
    do
    ...
  )
```

The user may optionally define an Analysis instance by providing the make, join, and modify functions.

As an example, let us implement a basic constant folding for our example language.

First, we define the analysis data structure:

```
structure AddMulData where
  data : Option Nat

instance : Inhabited (AddMulData) where
  default := {
    data := none
  }
```

It is required to define a default instance for the analysis. For our constant folding, we record an *Option Nat* per class; if the class holds a constant, then it holds a value of *some n* , and if it does not, it holds *none*.

As every analysis instance requires a make, join, and modify instance, we provide those definitions. Due to the modify instance requiring the make and join functions, it is often easier to define them separately as to not fall into a "cannot instantiate to instantiate" loop.

```
def myMake (en : ENode AddMul) (children : Array AddMulData) :
AddMulData :=
  -- Safe access for the analysis value
  let x := if children.isEmpty then
              (λ _   => none)
            else
              ((λ i  => (children[i]!).data) : (Nat → Option Nat))
  -- If a constant, lift to data
  -- If an operation, calculate if possible
  match en.head, en.args with
  | AddMul.lit n, _  => {data := some n}
  | .add, _          =>
    match x 0, x 1 with
    | some a, some b => {data := some (a + b)}
    | _     , _      => {data := none}
  | .mul, _          =>
    match x 0, x 1 with
    | some a, some b => {data := some (a * b)}
    | _     , _      => {data := none}
    | _, _           => {data := none}
```

```
def myJoin (d1 : AddMulData) (d2 : AddMulData) : AddMulData :=
  match d1.data, d2.data with
  | some v1 , some v2 =>
      if (v1 = v2) then
        d1
      else
        (panic! s!"Join fail, {v1} ≠ {v2}")
  | some _v1, _       => d1
  | _,       some _v2 => d2
  | _       , _       => {data := none}

instance ConstantFold : EGraph.Analysis AddMul AddMulData where

  make := myMake

  join := myJoin

  modify eg id :=
    let data := (eg.ecmap.get! id).data
    match data.data with
    | none => eg
    | some n =>
        let addNew : EGraphM AddMul AddMulData Unit := do
          let constId ← pushAnalysis (.lit n) myMake
          let _ ← union id constId myJoin
          return ()

        let (_, eg') := addNew.run eg
        eg'

abbrev AddMulIO := EGraphGenericIO AddMul AddMulData
```

We pass our language and analysis structure to EGraphGenericIO, which builds us our analysis e-graph state. In the next section we will see how to use the language to write and run tests.

Note: in our modify analysis we use the macro pushAnalysis instead of push, this is a wrapper around the internal push function.

5.2. **Writing Tests and Programs.**

An example test looks like.

```
def math_diff_same : MathLangIO Unit := do
  let lhs ← parseTerm "(d x x)"
  let rhs ← push (.const 1)
  eqSat (rules := mathRules) (limit := 1)
  let _ ← checkEquivalent lhs rhs
```

We run the pushes with the general format of *let id ← parseTerm "term"* , where the return value is the e-class id of the pushed term.

There are a few ways to creates nodes. For individual nodes, we can use the *push* macro and construct the node ourselves by passing the appropriate constructor. Similarly, a node with arguments can be pushed with *push .add [x,y]* where *x, y* denote the e-class ids of the arguments of the node.

For an s-expression, we can use the parser we defined along with our language and use the macro *parseTerm (+ 1 (+ 2 3))*, which recursively goes through the tree to push all the leaves (atoms) or lists needed to construct the expression.

Note: an expression is a string and must start with a parenthesis.

Therefore, these two sequences are equivalent.

```
Sequence 1:
let one ← push (.const 1)
let two ← push (.const 2)
let thr ← push (.const 3)
let p12 ← push .add [one, two]
let p3a ← push .add [thr, p12]

Sequence 2:
let p3a ← parseTerm (+ 3 (+ 1 2))
```

The difference being that in the first sequence, we have access to the e-class ids of all the arguments, while in the second sequence we only know the e-class id of the final term. However, thanks to the hashcons, it's perfectly possible to get the id of an individual node by pushing it again.

Finally, we also provide a checkEquivalent function that queries the e-graph whether two terms are syntactically equal.

At any time in the test, the user may call printEGraph to print the state of the e-graph.

An optional test_fn function is available in helpers.lean that mimics egg's test_fn! macro. This function takes a few other parameters that can help

```
def associativityTest : AddMul Unit := do
    test_fn
      (lhs :=  ("+ 1 (+ 2 (+ 3 (+ 4 (+ 5 (+ 6 7))))))"))
      (rhs := [("+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1))))))")])
      (rules := [
      r* pAdd (?"a") (?"b") === pAdd (?"b") (?"a"),
      r* pMul (?"a") (?"b") === pMul (?"b") (?"a"),
      r* pAdd (?"a") (pAdd (?"b") (?"c")) === pAdd (pAdd (?"a") (?"b"))
(?"c")
      ])

#eval runTest associativityTest
```

Other than an #eval block, defined tests can similarly be run in Main using the *runTest* function like follows:

```
def main : IO Unit := do
  let _ ← runTest associativityTest
```

We now take a look at our analysis. We again check for equality between (+ 3 (+ 2 1)) and (+ 1 (+ 2 3)), but print the e-graph state before and after saturation.

```
def quickerAssocAdd : AddMulIO Unit := do
let st ← parseTerm ("(+ 1 (+ 2 3))")

printEGraph
eqSat (limit := 3) (rules := simpleRules)
printEGraph
let rhs ← parseTerm ("(+ 3 (+ 2 1))")

let _ ← checkEquivalent st rhs

#eval runTest quickerAssocAdd
```

gives us

```
=== E-Graph State ===
Size: 5
ID 0 :: Nodes: #[{ head := AddMul.lit 1, args := #[] }]
ID 1 :: Nodes: #[{ head := AddMul.lit 2, args := #[] }]
ID 2 :: Nodes: #[{ head := AddMul.lit 3, args := #[] }]
ID 3 :: Nodes: #[{ head := AddMul.add, args := #[1, 2] }]
ID 4 :: Nodes: #[{ head := AddMul.add, args := #[0, 3] }]
====================

=== E-Graph State ===
Size: 22
ID 16 :: Nodes: #[{ head := AddMul.lit 4, args := #[] }, { head :=
AddMul.add, args := #[0, 2] },
  { head := AddMul.add, args := #[2, 0] }, { head := AddMul.add, args :=
#[1, 1] }]
ID 0 :: Nodes: #[{ head := AddMul.lit 1, args := #[] }]
ID 1 :: Nodes: #[{ head := AddMul.lit 2, args := #[] }, { head :=
AddMul.add, args := #[0, 0] }]
ID 2 :: Nodes: #[{ head := AddMul.lit 3, args := #[] }, { head :=
AddMul.add, args := #[0, 1] },
  { head := AddMul.add, args := #[1, 0] }]
ID 5 :: Nodes: #[{ head := AddMul.lit 5, args := #[] }, { head :=
AddMul.add, args := #[1, 2] },
  { head := AddMul.add, args := #[2, 1] }, { head := AddMul.add, args :=
#[16, 0] }]
ID 6 :: Nodes: #[{ head := AddMul.lit 6, args := #[] }, { head :=
AddMul.add, args := #[0, 5] },
  { head := AddMul.add, args := #[5, 0] }, { head := AddMul.add, args :=
#[2, 2] },
  { head := AddMul.add, args := #[16, 1] }, { head := AddMul.add, args :=
#[1, 16] }]
====================
```

We see that e-class 6 holds a node of *AddMul.lit 6*, as well as a node AddMul.add [0, 5], where classes 0 and 5 hold literals of 1 and 5 respectively. A quick glance at class 5 also shows a node AddMul.add [1, 2], where class 1 has *.lit 2* as well as .add [0, 0], and class 2 holds the constant 3. It seems the constant folding worked! We look at more examples in the next section

6. Tests and Evaluation

In this section, we try and evaluate the correctness and performance of our system by running some tests from egg(log).

6.1. **Correctness.**

For brevity's sake, we touch on each set of tests only shortly. The test files are available in the tests folder. Most tests do not take that long and can easily be run in the interpreter using *#eval*. However, tests that involve associativity and commutativity may take longer and it is suggested to compile and run.

Basic tests of operations on the e-graph are available in the file *egraphtests.lean*. These tests include:

- hash-consing
- congruence (define f(a) and f(b). If a and b are unioned, does f(a) ≡ f(b)?)
- propagation (union a b also propagates to g(f(a)) ≡ g(f(b))?)
- transitive (a ≡ b, b ≡ c. a ≡ c?)
- Tests were also run on random sequences of operations but have since been deleted..

Basic tests of rewriting on the e-graph are available in the file *rewritetests.lean*. Similar to the e-graph tests above, these simple tests focus on ensuring the core functionality works through basic operations. For most of the tests, we print the e-graph state and inspect it rather than asserting equivalence.

Egg's *prop.rs* tests were ported into *languages/prop.lean* and *tests/proptests.lean*. These tests include proving the contrapositive, chain, and constant folding.

```
Test Contrapositive
      Start: "(-> x y)"
      Assert Equivalence:
          "(-> x y)",
          "(| (~ x) y)",
          "(| (~ x) (~ (~ y)))",
          "(| (~ (~ y)) (~ x))",
          "(-> (~ y) (~ x))",
```

We attempt to port egg's *datalog.rs*. Some internal e-graph functions are used. Only one test was ported. The tests make use of conditional rewriting.

```
Base Case: If an edge (a, b) exists, and (a, b) is true, then (a, b) is a
path

Transitive: If a path (a, b) and an edge (b, c) exists, (a, c) is a path

Test: Edges (1,2) (2,3) and (3,4) are true. Find (1,4) is true and (4,1)
is false
```

Here we can see an example of conditional rewriting. As mentioned previously, a conditional rewrite takes a dictionary and the e-graph state monad to return a boolean.

```
def isTrueNode (head : Datalog) (v1 v2 : String) :
  Dict Datalog → EGraphM Datalog Unit Bool :=

  λ subst => do
    let id1 := subst.get! v1
    let id2 := subst.get! v2

    -- but we DON'T push it because we don't know if this exists or not!
    let enode : ENode Datalog := { head := head, args := #[id1, id2] }

    -- consult the hashcons
    match (← get).hcons.get? enode with
    | some id => isIdTrue id
    | none => return false
```

In this condition, we take the dictionary of matched values and get the ids of the pattern holes. We construct the node and check the hashcons for its id. In the case the node we want is an edge, we simply pass it on.

```
def isTrueEdge (a b : String) := isTrueNode .edge a b
```

This is used in building rules, such as

```
r* (pEdge (?"a") (?"b")) === (pPath (?"a") (?"b"))
   if (isTrueEdge "a" "b"),
```

where the edge (a, b) is only rewritten into a path (a, b) if it shares a class with Datalog.true_.

Finally, we port egg's *math.rs* in *languages/mathlang.lean* and *tests/mathtest.lean*. Here we hit a critical issue. The analysis instance for the test wants to do constant folding over arithmetic, but Lean4 does not support DecidableEq for float. As the exact analysis instance of rust cannot be implemented, we therefore change the type of the constant to Int. While this works for most, we make it so that division that does not lead to an integer invalidates the constant as to not conflict with each other. This does lead to a few tests breaking.

As the language is over a small calculus and arithmetic language, it is very similar to the AddMul language we used previously. There is not much of note, but there are plenty of examples that can be found in the testing file.

6.2. **Performance.**

Performance for each test was measured on a MacBook Air M2 with 16GB ram. The Lean version used is 4.27.0, and Rust is 1.91.1. The runtime of each test was measured in Rust with the command *cargo +nightly test – -Z unstable-options –test-threads=1 –report-time* to force serial execution. For Lean, *IO.monoMsNow* was called before and after each test to calculate the runtime.

Depending on the test case, the performance difference varies significantly. In particular, the math.rs test set showed significantly worse performance, with tests that complete in rust in 9ms took 225607ms in Lean, while a 4ms test took 63ms. Other tests have their memory usage balloon up to tens of gigabytes and fail to complete.

On other cases, like the prop.rs test set, the performance as measured shows no particular difference, with the times in ms both logging single digits. Curiously, for the constant folding prop test, egg logged 203ms while our implementation ran in 3ms.

As a pattern, the problem seems to get worse with associativity and commutativity. Therefore, the current working hypothesis is that egg, with its backoff scheduler and other heuristics, manages to avoid the e-graph growing uncontrollably, while our naïve system simply adds to the e-graph, leading to an exponential growth every iteration. It seems that by removing the offending rules, we could lead to a more controlled e-graph growth and reach saturation much faster. However, I am yet to explore this issue.

7. Limitation and Improvements

7.1. **Discussion.**

There are several limitations and inefficiencies in the system. One common inefficiency in the code is overcanonicalisation. When keying e-class ids, the code often canonicalises the keys before passing it to a function but the function also canonicalises the keys, resulting in redundant calculations. With an analysis of which functions call each other, we can reduce the amount of redundant calculations done.

Another obvious improvement is in the e-matching. Zhang et. al suggest that 60-90% of execution time is spent on e-matching [3], and the naïve backtracking that we implemented is unlikely to improve on that number. There are many alternative algorithms used in practice; egg uses the VM-based approach introduced by de Moura and Bjørner, and hegg, the Haskell port of Egg, uses relational e-matching. There are also improvements to the backtracking algorithm, suggested by Zhang in a blog post that could lead to major gains with minor changes. There are other novel techniques such as the one suggested by Krishnaswami and Yallop [4], that, to my understanding, build dedicated parser functions for different expressions. which Lean's capable metaprogramming seems to be a good fit for.

Finally, ideas from egg such as the backoff scheduler that controls heavily expansive rewrite rules look like they would help significantly in keeping the e-graph manageable. Egg also provides multiple conditions for terminating equality saturation, of which iteration limits and node limits were implemented, but terminating on goal would be helpful if there exists a desired stop condition, and time limits alongside the cost extraction would be useful for when a "good enough" optimisation is sufficient.

7.2. **Future Work.**

Clean up code and comments, add proper documentation. Section A in the Appendix does not yet contain all of the functions available to the user but will be updated to hold at least the ones that will be useful.

A quick thought is that a more efficient e-matching will still not handle an e-graph that grows too large, so the backoff scheduler might be of more importance than a better matching algorithm. Therefore finding some way to keep expansion in check would perhaps be of higher priority.

Some profiling of the code could help find if we're spending a disproportionate amount of time on a certain section, which could help target inefficiencies. This would be good to do before adding new features.

8. Reflection

I learned a lot about functional programming and e-graphs, and also quite a bit about how people are applying the idea to a wide range of domains. There were

a few things I could've done cleaner, in general a lot of the work I did was very unstructured and there are clear ways forward

If I had another month for the project, I would approach it in this order

1. Note a call graph of functions, try and reduce redundant behaviour as much as possible, as well as clean up code and factor into functions for readability. This can also help smooth out any bugs as I have to go through the code again.
2. Find some more test sets, without associativity/commutativity and do more extensive testing on rewrites. With these test sets, profile the code and compare to the profiling of other e-graph implementations to see if the amount of time we spend on each part is in the right ballpark. Use that to make an informed decision on priorities. We do this before any further changes to the code.
3. Implement that backoff scheduler, and then the priorities in the previous bullet point.

If I had to re-do the project from the beginning, I would take a much more structured approach and write down all we need first. While I tried to keep things such that every piece can be modified or replaced without any issues, often times I shuffled functionality between functions which led to lots of messy dependencies that needed to be ironed out. I would also take a much slower approach and test each change extensively before adding more. As it was, a lot of time was spent chasing and disabling the last few additions as it was often unclear which new changes were being problematic. I was also a bit impatient to extend as much functionality as possible, that I feel a lot of the code was neglected. Now that I have a better idea of the internal workings of the e-graph, perhaps it wouldn't be a bad time for a second try.

9. Conclusion

We built a simple, unoptimised e-graph that works over a generic language. While performance degrades significantly under certain conditions, it works for small languages and rewrite rules and provides a nice launching point to extend further.

References

1. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages. 5, 1–29 (2021). https://doi.org/10.1145/3434304
2. EGraphs-Good: Extraction Gym, https://github.com/egraphs-good/extraction-gym
3. Zhang, Y., Wang, Y.R., Willsey, M., Tatlock, Z.: Relational e-matching. Proc. ACM Program. Lang. 6, (2022). https://doi.org/10.1145/3498696
4. Krishnaswami, N.R., Yallop, J.: A typed, algebraic approach to parsing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 379–393. Association for Computing Machinery, Phoenix, AZ, USA (2019)

A: Functions Available to the User

A.1. Symbols and Abbreviations:

α is used to refer to the language defined by the user. There is a possibility this is changed to L in the future, but that should not affect functionality.

D is used to refer to the analysis data that can optionally be defined.

EClassId is an abbreviation for Nat

A.2. Useful Functions

ENode:
Definition:

```
structure ENode (α : Type _ ) where
  head : α
  args : Array EClassId
deriving Hashable, DecidableEq, Repr
```

EClass:
Definition:

```
structure EClass (α : Type _) (D : Type _) where
  nodes : Array (ENode α)
  parents : Array (ENode α × EClassId)
  data : D
deriving Repr
```

Constructors

```
def EClass.empty [Inhabited D] {α : Type _} : EClass α D
def EClass.fromNode {α : Type _} (en : ENode α) (data : D) : EClass α D
```

EGraph:
Definition:

```
structure EGraph (α : Type _) (D : Type _) [DecidableEq α] [Hashable α]
where
  uf    : Batteries.UnionFind
  ecmap : Std.HashMap EClassId (EClass α D)
  hcons : Std.HashMap (ENode α) EClassId
  dirty : Array EClassId
  opmap : Std.HashMap α (Array EClassId)

abbrev EGraphM (α : Type _) (D : Type _) [DecidableEq α] [Hashable α] :=
       StateM (EGraph α D)
```

Rewrite: While the functionality is no longer used, it is theoretically possible to run a single rewrite by doing `rewrite "your rule here"` , or `rewrite (#sym.alpha := "your language") (D := "your analysis") (r := "your rule")`.

Analysis:

Definition:

```
class Analysis (α : Type _) (D : Type _)  [DecidableEq α][Hashable α]
where
  make : (en : ENode α) → Array D → D
  join : D → D → D
  modify : EGraph α D → EClassId → EGraph α D
```

Helpers:

def lookupCanonicalEClassId (id : EClassId) : EGraphM α D <| EClassId
def findClass (en : ENode α) : EGraphM α D (Option EClassId)

Pushes the given node *en* and runs Analysis.make
def pushRun [Analysis α D] (en : ENode α) : EGraphM α D EClassId
Unions the given classes and runs Analysis.join
def unionRun [Analysis α D] (id_1 id_2 : EClassId) : EGraphM α D EClassId
Rebuilds the e-graph and runs Analysis.modify
def rebuildRun [Analysis α D] : EGraphM α D (Unit)

Conditional Rewriting: See section Section 5.1 for an example of how to use CustomLookup

```
inductive Condition (α : Type _) (D : Type _) [DecidableEq α] [Hashable α]
where
| Equal        : Pattern α → Pattern α           → Condition α D
| NotEqual     : Pattern α → Pattern α           → Condition α D
| CustomLookup : (Dict   α → EGraphM α D (Bool)) → Condition α D
```

Extraction:

```
abbrev cost := Nat

abbrev costFn α := α → cost

partial def extract (eg : EGraph α D) (fn : costFn α) : costMap α
```