Egg: An Equality Saturation Tactic in Lean

Marcus Rossel Barkhausen Institut
Andrés Goens University of Amsterdam
Rudi Schneider Technische Universität Berlin

Recently on Lean Together 2025

```
set_option diagnostics true
```

```
def f (x : Nat) := x + 1
def g (x : Nat) := 1 + x

@[simp] theorem f_eq : f x = g x := by simp_arith [f, g]
@[simp] theorem g_eq : g x = f x := by simp_arith [f, g]

example : f (x + 1) = x + 2 := by
  set_option diagnostics true in
  simp
```

```
[simp] used theorems (max: 249, num: 2): ▼
  f_eq  @f_eq : ∀ {x : Nat}, f x = g x
  g_eq ↦ 249
[simp] tried theorems (max: 250, num: 2): ▶
use `set_option diagnostics.threshold <num>` to control threshold for reporting counters

▼simp_issue.lean:19:2

tactic 'simp' failed, nested error:
maximum recursion depth has been reached
use `set_option maxRecDepth <num>` to increase limit
use `set_option diagnostics true` to get diagnostic information
```

What if `simp` fails?

```
def f (x : Nat) := x + 1
def g (x : Nat) := 1 + x

@[simp] theorem f_eq : f x = g x := by simp_arith [f, g]
@[simp] theorem g_eq : g x = f x := by simp_arith [f, g]

example: f (x + 1) = x + 2 := by
  simp
```

What if simp fails?

```
def f (x : Nat) := x + 1
def g (x : Nat) := 1 + x

@[simp] theorem f_eq : f x = g x := by simp_arith [f, g]
@[simp] theorem g_eq : g x = f x := by simp_arith [f, g]

example: f (x + 1) = x + 2 := by
  simp

example: f (x + 1) = x + 2 := by
  rw [f, g, f_eq, g_eq]
```

What if simp fails?

```
def f (x : Nat) := x + 1
def g (x : Nat) := 1 + x

@[simp] theorem f_eq : f x = g x := by simp_arith [f, g]
@[simp] theorem g_eq : g x = f x := by simp_arith [f, g]

example: f (x + 1) = x + 2 := by
  simp

example: f (x + 1) = x + 2 := by
  rw [f, g, f_eq, g_eq]

example: f (x + 1) = x + 2 := by
  rw [f]
```

What if `rw` is annoying?

What if rw is annoying?

```
variable [Group G] {a b : G}

example : (1 : G)⁻¹ = 1 := by
  simp only [mul_assoc, one_mul, mul_one, inv_mul_cancel, mul_inv_cancel]
```

What if rw is annoying?

```
variable [Group G] {a b : G}

example : (1 : G)⁻¹ = 1 := by
  simp only [mul_assoc, one_mul, mul_one, inv_mul_cancel, mul_inv_cancel]

example : (1 : G)⁻¹ = 1 := by
  rw [← mul_one, inv_mul_cancel]
```

What if rw is annoying?

```
variable [Group G] {a b : G}

example : (1 : G)⁻¹ = 1 := by
  simp only [mul_assoc, one_mul, mul_one, inv_mul_cancel, mul_inv_cancel]

example : (1 : G)⁻¹ = 1 := by
  rw [← mul_one, inv_mul_cancel]

example : (1 : G)⁻¹ = 1 := by
  rw [← mul_one 1⁻¹, inv_mul_cancel]
```

What if rw is annoying?

```
variable [Group G] {a b : G}

example : (1 : G)⁻¹ = 1 := by
  simp only [mul_assoc, one_mul, mul_one, inv_mul_cancel, mul_inv_cancel]

example : (1 : G)⁻¹ = 1 := by
  rw [← mul_one, inv_mul_cancel]

example : (1 : G)⁻¹ = 1 := by
  rw [← mul_one 1⁻¹, inv_mul_cancel]

example : (1 : G)⁻¹ = 1 := by
  group
```

What if there is no tactic for your domain?

What if there is no tactic for your domain?

```
@[simp]
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  rw [← sub_eq_zero, sub_neg_eq_add, ← add_lie]
  simp
```

What if there is no tactic for your domain?

```
@[simp]
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  rw [← sub_eq_zero, sub_neg_eq_add, ← add_lie]
  simp
```

```
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  egg [sub_eq_zero, sub_neg_eq_add, add_lie, neg_add_cancel, zero_lie]
```

What if there is no tactic for your domain?

```
@[simp]
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  rw [← sub_eq_zero, sub_neg_eq_add, ← add_lie]
  simp
```

```
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  egg [sub_eq_zero, sub_neg_eq_add, add_lie, neg_add_cancel, zero_lie]
```

```
theorem inv_one : (1 : G)⁻¹ = 1 := by
  egg [mul_assoc, one_mul, mul_one, inv_mul_cancel, mul_inv_cancel]
```

Tactic Comparsion

`rw`

`egg`

`simp`

Tactic Comparsion

`rw`	`egg`	`simp`
based on rewriting	based on rewriting	based on rewriting

Tactic Comparsion

`rw`	`egg`	`simp`
based on rewriting	based on rewriting	based on rewriting
local knowledge	domain-specific knowledge	global knowledge

Tactic Comparsion

`rw`	`egg`	`simp`
based on rewriting	based on rewriting	based on rewriting
local knowledge	domain-specific knowledge	global knowledge
non-/terminal	terminal *	non-/terminal

Tactic Comparsion

rw	egg	simp
based on rewriting	based on rewriting	based on rewriting
local knowledge	domain-specific knowledge	global knowledge
non-/terminal	terminal *	non-/terminal
really fast	slow	fast

Tactic Comparsion

`rw`	`egg`	`simp`
based on rewriting	based on rewriting	based on rewriting
local knowledge	domain-specific knowledge	global knowledge
non-/terminal	terminal *	non-/terminal
really fast	slow	fast
manual rewriting	equality saturation	greedy rewriting

Equality Saturation

Equality Saturation: a New Approach to Optimization *

Ross Tate Michael Stepp Zachary Tatlock Sorin Lerner
Department of Computer Science and Engineering
University of California, San Diego
{rtate, mstepp, ztatlock, lerner} @cs.ucsd.edu

Abstract

Optimizations in a traditional compiler are applied sequentially, with each optimization destructively modifying the program to produce a transformed program that is then passed to the next optimization. We present a new approach for structuring the optimization phase of a compiler. In our approach, optimizations take the form of equality analyses that add equality information to a common intermediate representation. The optimizer works by repeatedly applying these analyses to infer equivalences between program fragments, thus saturating the intermediate representation with equalities. Once saturated, the intermediate representation encodes multiple optimized versions of the input program. At this point, a profitability heuristic picks the final optimized program from the various programs represented in the saturated representation. Our proposed way of structuring optimizers has a variety of benefits over previous approaches: our approach obviates the need to worry about optimization ordering, enables the use of a global optimization heuristic that selects among fully optimized programs, and can be used to perform translation validation, even on compilers other than our own. We present our approach, formalize it, and describe our choice of intermediate representation. We also present experimental results showing that our approach is practical in terms of time and space overhead, is effective at discovering intricate optimization opportunities, and is effective at performing translation validation for a realistic optimizer.

generated code, a problem commonly known as the *phase ordering problem*. Another drawback is that profitability heuristics, which decide whether or not to apply a given optimization, tend to make their decisions one optimization at a time, and so it is difficult for these heuristics to account for the effect of future transformations.

In this paper, we present a new approach for structuring optimizers that addresses the above limitations of the traditional approach, and also has a variety of other benefits. Our approach consists of computing a set of optimized versions of the input program and then selecting the best candidate from this set. The set of candidate optimized programs is computed by repeatedly inferring equivalences between program fragments, thus allowing us to represent the effect of many possible optimizations at once. This, in turn, enables the compiler to delay the decision of whether or not an optimization is profitable until it observes the full ramifications of that decision. Although related ideas have been explored in the context of super-optimizers, as Section 8 on related work will point out, super-optimizers typically operate on straight-line code, whereas our approach is meant as a general-purpose compilation paradigm that can optimize complicated control flow structures.

At its core, our approach is based on a simple change to the traditional compilation model: whereas traditional optimizations operate by destructively performing transformations, in our approach optimizations take the form of *equality analyses* that simply add equality information to a common intermediate representation (IR), without losing the original program. Thus, after each equality anal-

Equality Saturation: Phase Ordering Problem

Equality Saturation: Phase Ordering Problem

Program

Optimizations

Equality Saturation: Phase Ordering Problem

Program

$$\frac{a \cdot 2}{2}$$

Optimizations

$$x \cdot 2 \implies x \ll 1$$

$$\frac{x \cdot y}{z} \implies x \cdot \frac{y}{z}$$

$$\frac{x}{x} \implies 1$$

$$x \cdot 1 \implies x$$

Equality Saturation: Phase Ordering Problem

Program

$$\frac{a \cdot 2}{2} \implies \frac{a \ll 1}{2} \implies$$ 

Optimizations

$$x \cdot 2 \implies x \ll 1$$

$$\frac{x \cdot y}{z} \implies x \cdot \frac{y}{z}$$

$$\frac{x}{x} \implies 1$$

$$x \cdot 1 \implies x$$

Equality Saturation: Phase Ordering Problem

Program

$$\frac{a \cdot 2}{2} \Longrightarrow \frac{a \ll 1}{2} \Longrightarrow$$

$$\Downarrow$$

$$a \cdot \frac{2}{2} \Longrightarrow a \cdot 1 \Longrightarrow a$$

Optimizations

$$x \cdot 2 \Longrightarrow x \ll 1$$

$$\frac{x \cdot y}{z} \Longrightarrow x \cdot \frac{y}{z}$$

$$\frac{x}{x} \Longrightarrow 1$$

$$x \cdot 1 \Longrightarrow x$$

Equality Saturation: Phase Ordering Problem

Program

$$\frac{a \cdot 2}{2} \implies \frac{a \ll 1}{2} \implies$$

$$\Downarrow$$

$$a \cdot \frac{2}{2} \implies a \cdot 1 \implies a$$

Optimizations

$$x \cdot 2 \implies x \ll 1$$

$$\frac{x \cdot y}{z} \implies x \cdot \frac{y}{z}$$

$$\frac{x}{x} \implies 1$$

$$x \cdot 1 \implies x$$

Phase Ordering Problem

In which order should we apply optimizations?

Equality Saturation: Phase Ordering Problem

Proof Goal

$$\frac{a \cdot 2}{2} \implies \frac{a \ll 1}{2} \implies$$

$$\Downarrow$$

$$a \cdot \frac{2}{2} \implies a \cdot 1 \implies a$$

Equational Theorems

$$x \cdot 2 \implies x \ll 1$$

$$\frac{x \cdot y}{z} \implies x \cdot \frac{y}{z}$$

$$\frac{x}{x} \implies 1$$

$$x \cdot 1 \implies x$$

Phase Ordering Problem

In which order should we apply optimizations?

Equality Saturation: Phase Ordering Problem

Proof Goal

$$\frac{a \cdot 2}{2} \Longrightarrow \frac{a \ll 1}{2} \Longrightarrow$$

$$\Downarrow$$

$$a \cdot \frac{2}{2} \Longrightarrow a \cdot 1 \Longrightarrow a$$

Equational Theorems

$$x \cdot 2 \Longrightarrow x \ll 1$$

$$\frac{x \cdot y}{z} \Longrightarrow x \cdot \frac{y}{z}$$

$$\frac{x}{x} \Longrightarrow 1$$

$$x \cdot 1 \Longrightarrow x$$

Phase Ordering Problem

In which order should we apply optimizations?

“Solution“

Try all possible orders.

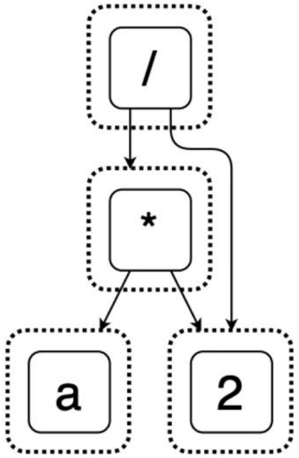
Equality Saturation: E-Graphs

Equality Saturation: E-Graphs

E-Graph ≈ Term Graph + Congruence Relation

Equality Saturation: E-Graphs

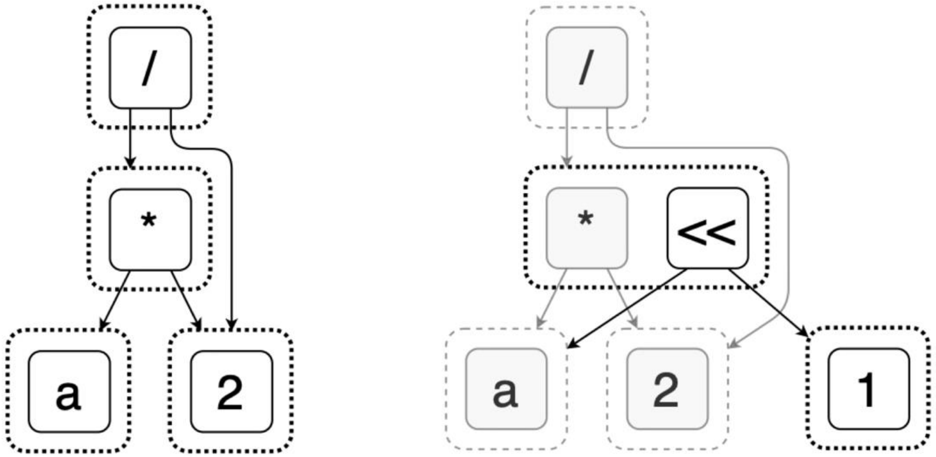
E-Graph ≈ Term Graph + Congruence Relation



(a) Initial e-graph contains $(a \times 2)/2$.

Equality Saturation: E-Graphs

E-Graph ≈ Term Graph + Congruence Relation

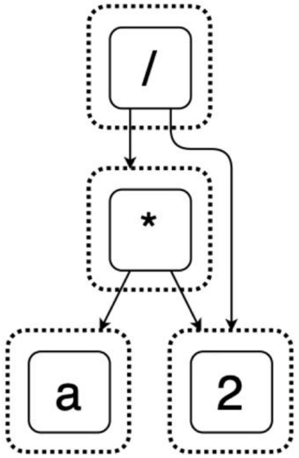


(a) Initial e-graph contains $(a \times 2)/2$.

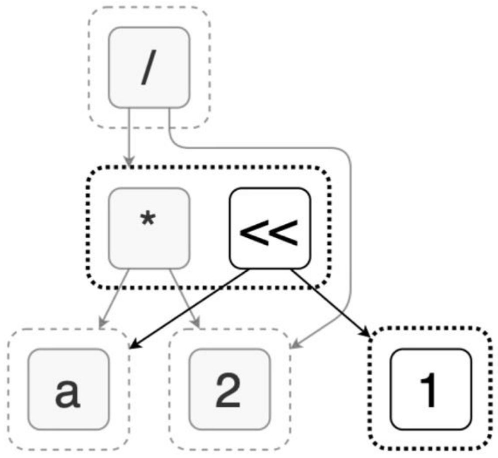
(b) After applying rewrite $x \times 2 \to x \ll 1$.

Equality Saturation: E-Graphs

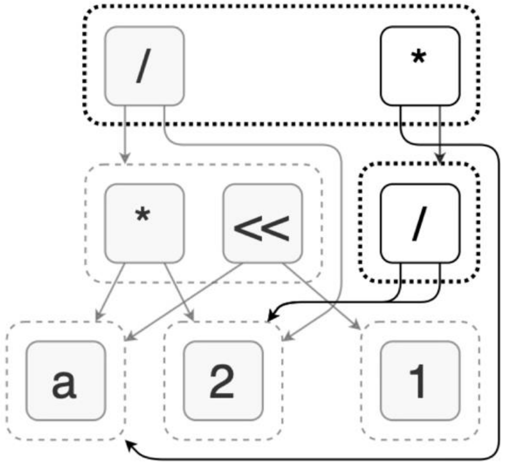
E-Graph ≈ Term Graph + Congruence Relation



(a) Initial e-graph contains $(a \times 2)/2$.



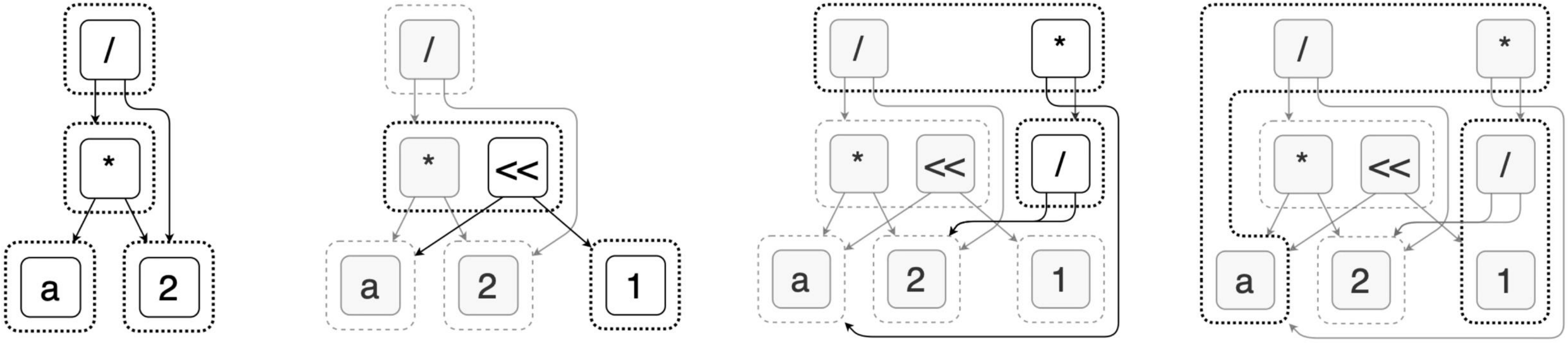
(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.



(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

Equality Saturation: E-Graphs

E-Graph ≈ Term Graph + Congruence Relation



(a) Initial e-graph contains $(a \times 2)/2$.

(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

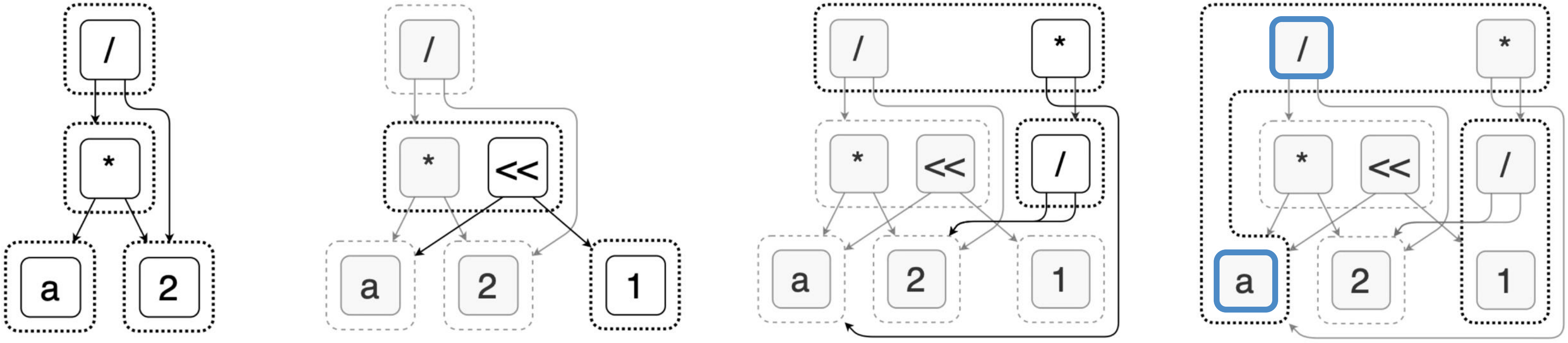
(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Equality Saturation: E-Graphs

E-Graph ≈ Term Graph + Congruence Relation

$$\frac{a \cdot 2}{2} = a$$



(a) Initial e-graph contains $(a \times 2)/2$.

(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Previous Work

EGRAPHS 2022

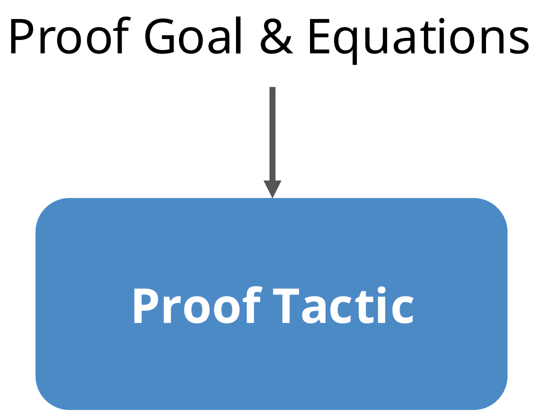
☆ Equality Saturation as a Tactic for Proof Assistants

Rewrites are an essential component of proof assistants. Term rewrite systems are well-studied methods to deal with these kinds of rewrites in a formal setting. A limitation of arbitrary term rewrite systems is the destructive nature of rewrites. In contrast, in Egraphs, applying a rewrite also keeps the previous representative. In a sense, this applies the rewrite in both directions. The main idea of this talk is using equality saturation in proof assistants as a powerful tactic, i.e. a meta-program to partially automate the task of finding proofs. We will discuss the limitations of Egraph-based rewrites and our proposed solutions to these. We do this using the Lean proof assistant and the egg framework, considering examples from group theory.

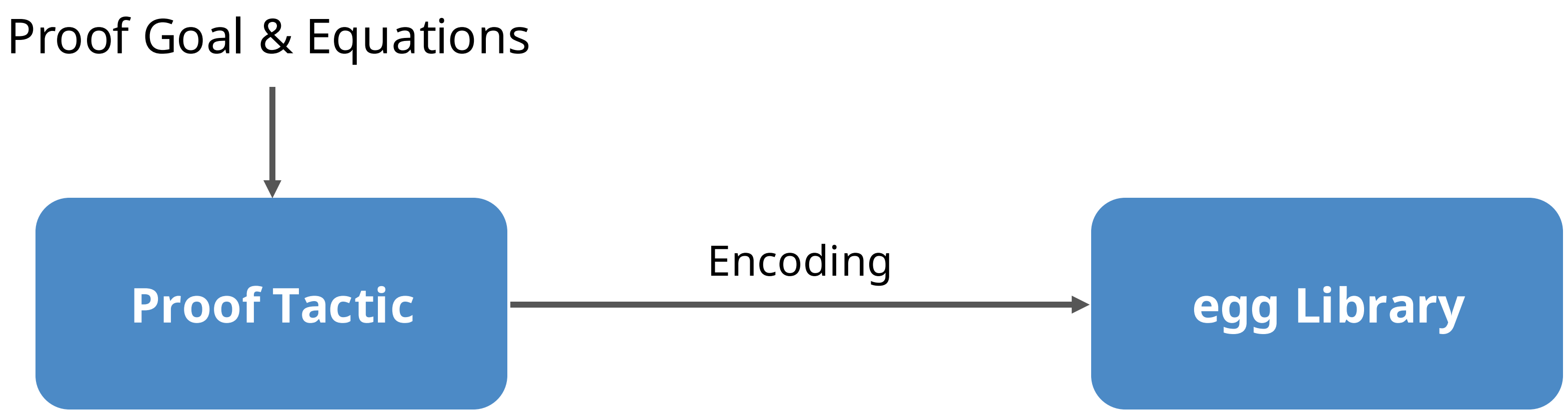
Andrés Goens
the University of Edinburgh

Siddharth Bhat
the University of Edinburgh

Basic Pipeline



Basic Pipeline



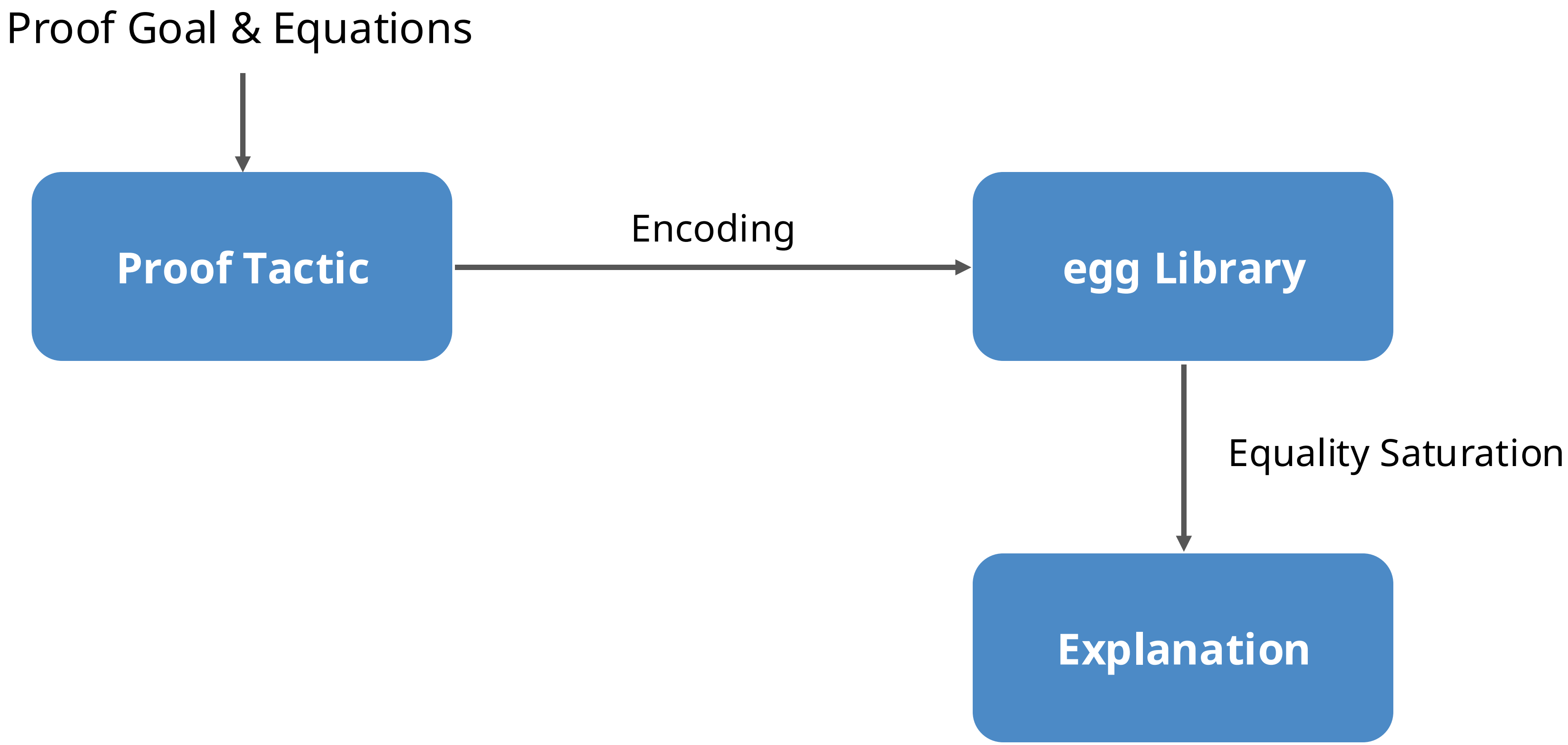
Encoding

```
set_option trace.egg.encoded true in
example : ⁅-x, m⁆ = -⁅x, m⁆ := by
  egg [neg_add_cancel, zero_lie, sub_eq_zero, sub_neg_eq_add, add_lie]
```

Lean Infoview

```
[egg.encoded] Encoded ▼
  [] Goal ▼
  [] LHS: (app (app (app (app (app (const "Bracket.bracket" (param "v") (param "w")) (fvar 13281))
(fvar 13283)) (inst (app (app (const "Bracket" (param "v") (param "w")) (fvar 13281)) (fvar 13283))))
(app (app (app (const "Neg.neg" (param "v")) (fvar 13281)) (inst (app (const "Neg" (param "v")) (fvar
13281)))) (fvar 13897))) (fvar 13903))
  [] RHS: (app (app (app (const "Neg.neg" (param "w")) (fvar 13283)) (inst (app (const "Neg" (param
"w")) (fvar 13283)))) (app (app (app (app (app (const "Bracket.bracket" (param "v") (param "w")) (fvar
13281)) (fvar 13283)) (inst (app (app (const "Bracket" (param "v") (param "w")) (fvar 13281)) (fvar
13283)))) (fvar 13897)) (fvar 13903)))
```

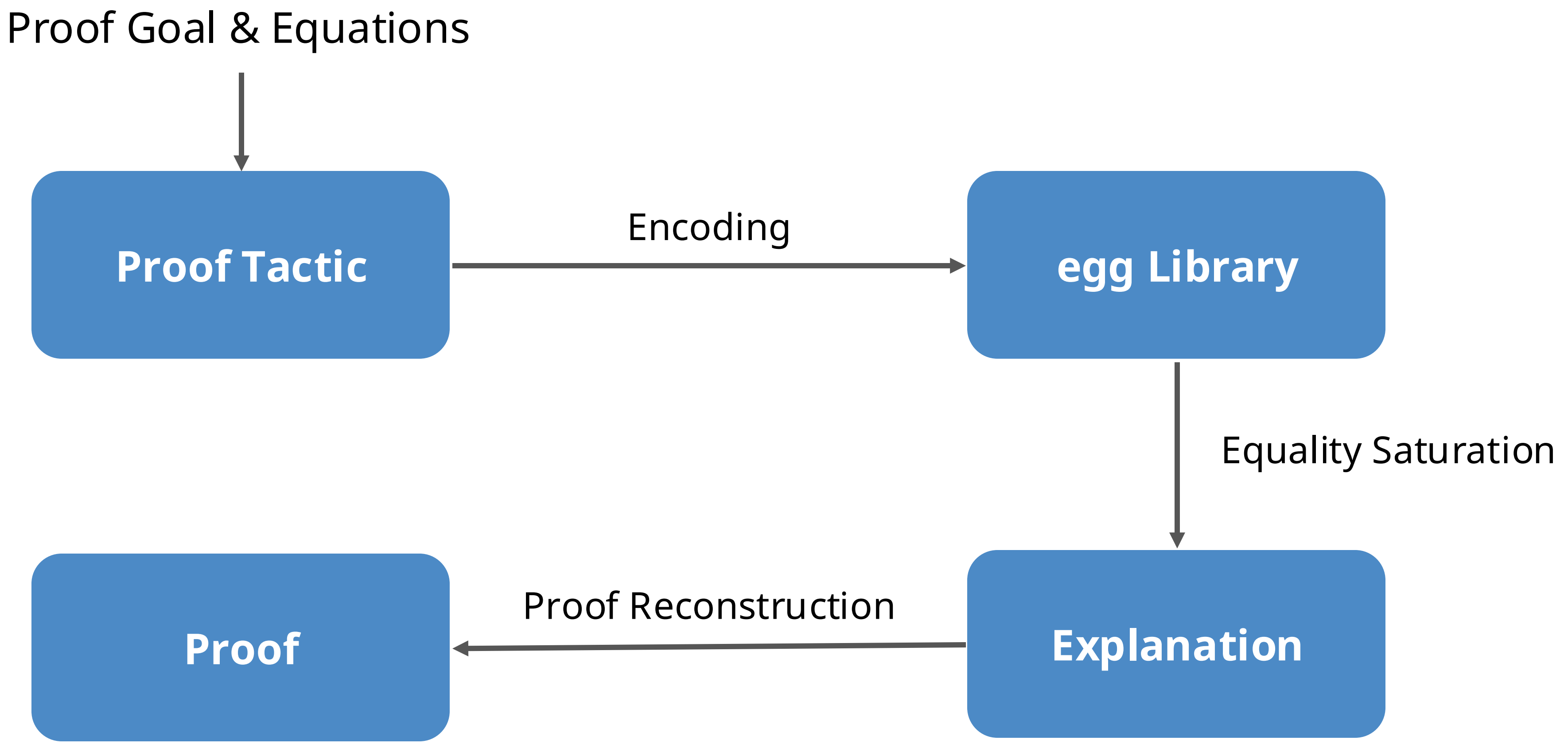
Basic Pipeline



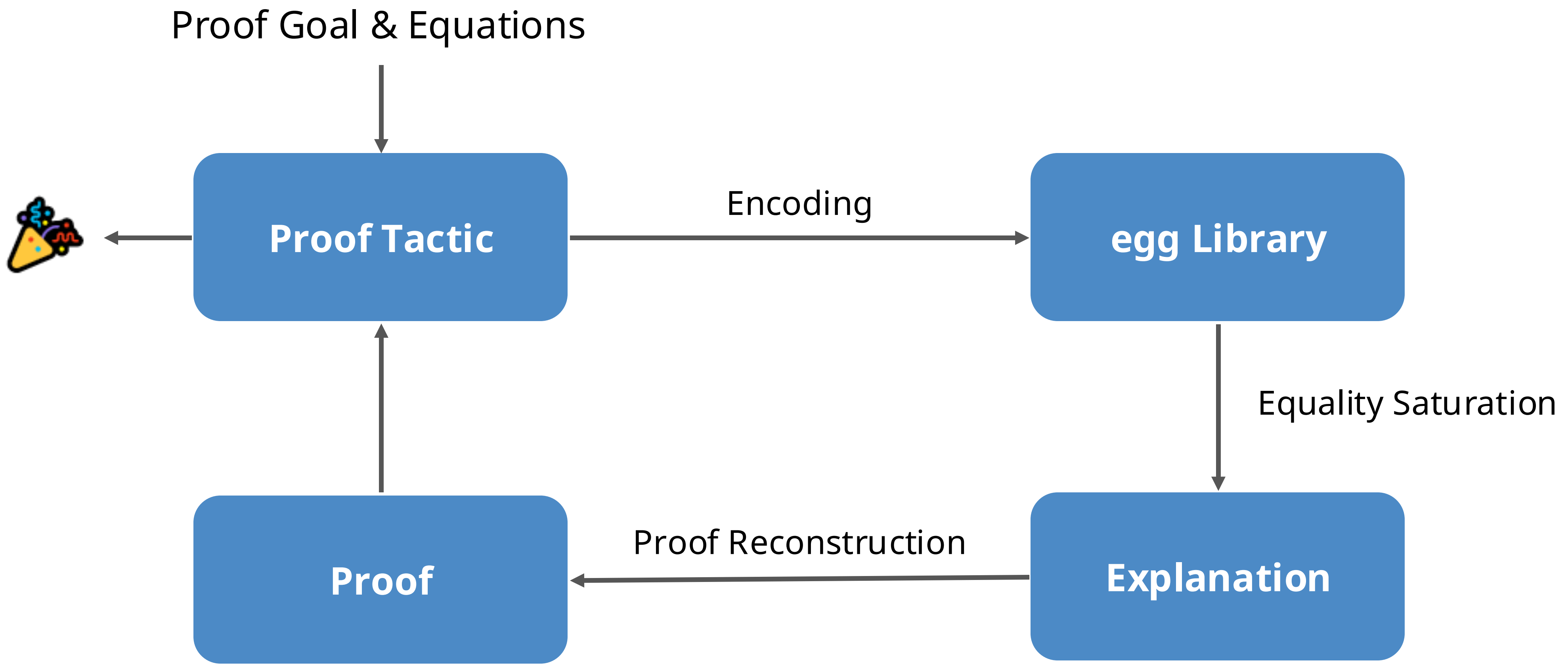
Explanations

```
(= (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903)) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (Rewrite=> #2<⊢0>-rev (= (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (app (app (app (const OfNat.ofNat (param w)) (fvar 13283)) (lit 0)) (inst (app (app (const OfNat (param w)) (fvar 13283)) (lit 0)))))) (= (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (Rewrite<= #1 (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const OfNat.ofNat (param v)) (fvar 13281)) (lit 0)) (inst (app (app (const OfNat (param v)) (fvar 13281)) (lit 0))))) (fvar 13903)))) (= (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (Rewrite<= #0 (app (app (app (app (app (app (const HAdd.hAdd (param v) (param v) (param v)) (fvar 13281)) (fvar 13281)) (fvar 13281)) (inst (app (app (app (const HAdd (param v) (param v) (param v)) (fvar 13281)) (fvar 13281)) (fvar 13281)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13897)))) (fvar 13903))) (= (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (Rewrite=> #4 (app (app (app (app (app (app (const HAdd.hAdd (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HAdd (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903))))) (= (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (Rewrite<= #3 (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))))) (= (app (app (Rewrite=> #2<⊢0>[◂69632,0] (app (app (const Sub.sub (param w)) (fvar 13283)) (inst (app (const Sub (param w)) (fvar 13283))))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (app (app (app (app (app (app (const HSub.hSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)) (inst (app (app (app (const HSub (param w) (param w) (param w)) (fvar 13283)) (fvar 13283)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903))))) (= (app (app (app (app (const Sub.sub (param w)) (fvar 13283)) (inst (app (const Sub (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903)))) (app (app (Rewrite=> #2<⊢0>[◂69632,0] (app (app (const Sub.sub (param w)) (fvar 13283)) (inst (app (const Sub (param w)) (fvar 13283))))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (app (app (app (const Neg.neg (param v)) (fvar 13281)) (inst (app (const Neg (param v)) (fvar 13281)))) (fvar 13897))) (fvar 13903))) (app (app (app (const Neg.neg (param w)) (fvar 13283)) (inst (app (const Neg (param w)) (fvar 13283)))) (app (app (app (app (app (const Bracket.bracket (param v) (param w)) (fvar 13281)) (fvar 13283)) (inst (app (app (const Bracket (param v) (param w)) (fvar 13281)) (fvar 13283)))) (fvar 13897)) (fvar 13903))))) (Rewrite=> = (const True))
```

Basic Pipeline



Basic Pipeline



Proof

```
set_option trace.egg.explanation true in
example : ⁅-x, m⁆ = -⁅x, m⁆ := by
  egg [neg_add_cancel, zero_lie, sub_eq_zero, sub_neg_eq_add, add_lie]
```

Lean Infoview

```
[egg.explanation.steps] Explanation Steps ▼
  [] ⁅-x, m⁆ = -⁅x, m⁆
  [] 0: ⁅-x, m⁆ - -⁅x, m⁆ = 0 ▶
  [] 1: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅0, m⁆ ▶
  [] 2: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x + x, m⁆ ▶
  [] 3: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x, m⁆ + ⁅x, m⁆ ▶
  [] 4: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x, m⁆ - -⁅x, m⁆ ▶
  [] 5: Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) = ⁅-x, m⁆ - -⁅x, m⁆ ▶
  [] 6: Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) = Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) ▶
  [] 7: True ▶
```

Proof

```
set_option trace.egg.explanation true in
example : ⁅-x, m⁆ = -⁅x, m⁆ := by
  egg [neg_add_cancel, zero_lie, sub_eq_zero, sub_neg_eq_add, add_lie]

```

Lean Infoview

```
[egg.explanation.steps] Explanation Steps ▼
  [] ⁅-x, m⁆ = -⁅x, m⁆
  [] 0: ⁅-x, m⁆ - -⁅x, m⁆ = 0 ▶
  [] 1: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅0, m⁆ ▶
  [] 2: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x + x, m⁆ ▶
  [] 3: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x, m⁆ + ⁅x, m⁆ ▶
  [] 4: ⁅-x, m⁆ - -⁅x, m⁆ = ⁅-x, m⁆ - -⁅x, m⁆ ▶
  [] 5: Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) = ⁅-x, m⁆ - -⁅x, m⁆ ▶
  [] 6: Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) = Sub.sub ⁅-x, m⁆ (-⁅x, m⁆) ▶
  [] 7: True ▶
```

¯_(ツ)_/¯

Guides

1

Guided Equality Saturation

THOMAS KŒHLER, Inria & Université de Strasbourg, France
ANDRÉS GOENS, University of Edinburgh, UK
SIDDHARTH BHAT, University of Edinburgh, UK
TOBIAS GROSSER, University of Edinburgh, UK
PHIL TRINDER, University of Glasgow, UK
MICHEL STEUWER, University of Edinburgh, UK

Rewriting is a powerful and principled term transformation technique with uses across theorem proving and compilation. In theorem proving, each rewrite is a proof step; in compilation, rewrites optimize a program term. While developing rewrite sequences manually is possible, this process does not scale when larger rewrite sequences are needed. Automated rewriting techniques, like greedy simplification or equality saturation, work well without requiring human input. Yet, they do not scale to large and complex search spaces, which limits the complexity of tasks where automated rewriting is effective, and means that just a small increase in term size or rewrite sequence length may result in failure.

This paper proposes a semi-automatic rewriting technique as a means to scale rewriting by allowing for human input at key decision points. Specifically, we propose *guided equality saturation* that embraces human guidance when fully automated equality saturation does not scale. A human provides an intermediate *guide*, and the rewriting is split into two simpler automatic equality saturation steps: from the original term to the guide, and from the guide to the target. A complex rewriting task may require multiple guides, resulting in a sequence of equality saturation steps. A guide need not be a complete term, it can also be a *sketch* containing undefined elements that are instantiated by the equality saturation search. Such sketches may be much more concise than complete program terms.

We demonstrate the generality and effectiveness of guided equality saturation using case studies in theorem proving and program optimization. First, we introduce guided equality saturation as a novel tactic in the Lean 4 proof assistant, allowing proofs to be written in the style of textbook proof sketches, i.e., as a series of calculations that omit details and skip steps. This tactic concludes in fractions of a second instead of minutes when compared to unguided equality saturation, and can find complex proofs that previously had to be done manually. Second, in the compiler of the RISE functional array language, where unguided equality saturation fails to perform advanced optimizations within an hour and using 60 GB of memory, guided equality saturation performs the same optimizations with up to 3 guides, within seconds and using less than 1 GB of memory.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**; **Automated reasoning**; • **Software and its engineering → Compilers**; • **General and reference → Performance**.

Guides

```
/-- The inverse of a bijective morphism is a morphism. -/
def inverse (f : L₁ →ₗ⁅R⁆ L₂) (g : L₂ → L₁) (h₁ : Function.LeftInverse g f)
    (h₂ : Function.RightInverse g f) : L₂ →ₗ⁅R⁆ L₁ :=
  { LinearMap.inverse f.toLinearMap g h₁ h₂ with
    map_lie' := by
      intros x y
      calc
        g ⁅x, y⁆ = g ⁅f (g x), f (g y)⁆ := by conv_lhs => rw [← h₂ x, ← h₂ y]
        _ = g (f ⁅g x, g y⁆) := by rw [map_lie]
        _ = ⁅g x, g y⁆ := h₁ _
        }
```

Guides

```
/-- The inverse of a bijective morphism is a morphism. -/
def inverse (f : L₁ →ₗ⁅R⁆ L₂) (g : L₂ → L₁) (h₁ : Function.LeftInverse g f)
    (h₂ : Function.RightInverse g f) : L₂ →ₗ⁅R⁆ L₁ :=
  { LinearMap.inverse f.toLinearMap g h₁ h₂ with
    map_lie' := by
      intros x y
      calc
        g ⁅x, y⁆ = g ⁅f (g x), f (g y)⁆ := by conv_lhs => rw [← h₂ x, ← h₂ y]
        _ = g (f ⁅g x, g y⁆) := by rw [map_lie]
        _ = ⁅g x, g y⁆ := h₁ _
         }
```

Guides

```
/-- The inverse of a bijective morphism is a morphism. -/
def inverse (f : L₁ →ₗ⁅R⁆ L₂) (g : L₂ → L₁) (h₁ : Function.LeftInverse g f)
    (h₂ : Function.RightInverse g f) : L₂ →ₗ⁅R⁆ L₁ :=
  { LinearMap.inverse f.toLinearMap g h₁ h₂ with
    map_lie' := by
      intros x y
      egg calc [h₂ x, h₂ y, map_lie, h₁ _]
        g ⁅x, y⁆ = g ⁅f (g x), f (g y)⁆
        _ = g (f ⁅g x, g y⁆)
        _ = ⁅g x, g y⁆
         }
```

Guides

$$
\begin{aligned}
&\frac{n!}{(r-1)!(n-r+1)!} + \frac{n!}{r!(n-r)!} \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{1}{n-r+1} + \frac{1}{r}\right) \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{r+n-r+1}{r(n-r+1)}\right) \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{n+1}{r(n-r+1)}\right) \\
&= \frac{(n+1)!}{r!(n+r-1)!}
\end{aligned}
$$

Guides

$$
\begin{aligned}
&\frac{n!}{(r-1)!(n-r+1)!}+\frac{n!}{r!(n-r)!}\\
&=\frac{n!}{(r-1)!(n-r)!}\left(\frac{1}{n-r+1}+\frac{1}{r}\right)\\
&=\frac{n!}{(r-1)!(n-r)!}\left(\frac{r+n-r+1}{r(n-r+1)}\right)\\
&=\frac{n!}{(r-1)!(n-r)!}\left(\frac{n+1}{r(n-r+1)}\right)\\
&=\frac{(n+1)!}{r!(n+r-1)!}
\end{aligned}
$$

```
egg calc [add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
          mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
          div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
  _ = (n ! / ((r - 1) ! * (n - r + 1) !) + n ! / (r ! * (n - r) !))
  _ = n ! / ((r - 1) ! * (n - r) !) * (1 / (n - r + 1) + 1 / r)
  _ = n ! / ((r - 1) ! * (n - r) !) * ((r + (n - r + 1)) / (r * (n - r + 1)))
  _ = n ! / ((r - 1) ! * (n - r) !) * ((n + 1) / (r * (n - r + 1)))
  _ = (n + 1) ! / (r ! * (n + 1 - r) !)
```

Future Work

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Future Work

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Proof obligations for conditional rewrites

Future Work

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Proof obligations for conditional rewrites should be an output of egg

Future Work

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Future Work

Simp Set

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Future Work

~~Simp Set~~

Egg Basket *Thanks to Johan Commelin*

```
[add_comm, sub_add_cancel, sub_add_eq_add_sub, mul_one, mul_comm, mul_assoc,
 mul_div_mul_left, _root_.div_mul_div_comm, _root_.add_div, left_distrib,
 div_mul_eq_div_mul_one_div, Real.Gamma_add_one, h₄, h₅, h₆]
```

Future Work

~~Simp Set~~

Egg Basket

Thanks to Johan Commelin

```
egg lie
egg real
egg group
egg list
```

Future Work

Bundle the Backend in Lake Project

Proof Persistence

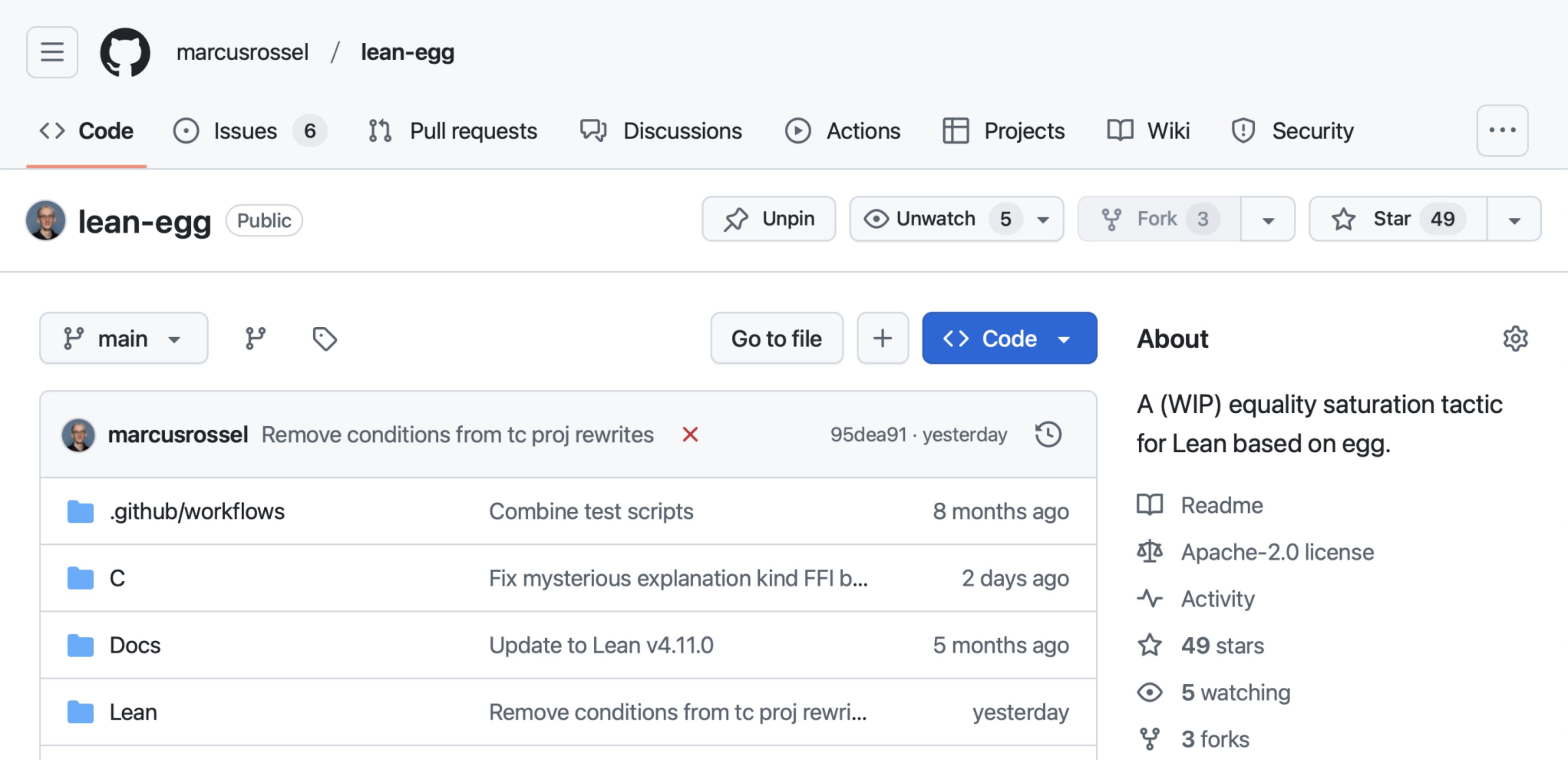
Egg Basket

Congruence Theorems in Proof Reconstruction

Output Proof Obligations for Conditional Rewrites

Premise Selection

Try it out… at your own risk.



Questions?