Proof Primitives for Equality Saturation-based Automated Provers

George Zakhour
University of St. Gallen
St. Gallen, Switzerland
george.zakhour@unisg.ch

Jahrim Gabriele Cesario
University of St. Gallen
St. Gallen, Switzerland
jahrimgabriele.cesario@unisg.ch

Pascal Weisenburger
University of St. Gallen
St. Gallen, Switzerland
pascal.weisenburger@unisg.ch

Guido Salvaneschi
University of St. Gallen
St. Gallen, Switzerland
guido.salvaneschi@unisg.ch

Abstract

We present a proof extraction algorithm for versioned e-graphs, an extension of e-graphs that supports branching reasoning contexts and proof by cases. In the setting of algebraic datatypes, the algorithm handles rewrites, congruence, injectivity, contradictions, induction, and case splits. We implement it in Vegie, a lightweight automated inductive theorem prover. An early case study suggests that the produced proofs are smaller than those of a state-of-the-art e-graph prover.

Keywords: Proof Extraction, E-Graph, Equality Saturation

1 Introduction

The power of automated provers comes from efficiently navigating a large space of proofs, i.e., terms that are equivalent and can be rewritten into one another. To do so, tools follow one of two strategies. Either they apply all rewrites to all terms [20], keeping track of these choices in a wide and deep search tree. Or they develop heuristics [17] or machine-learning techniques [4] to prune out the search space and favour an order of rewrites over others. The first strategy is obviously vulnerable to state explosion, while the second is vulnerable to blind spots in the search space. Therefore, in practice, a remedy is to mix both strategies: to implement some heuristics while multiple branches are explored simultaneously.

In all cases, automated provers must be able to store and manipulate a large amount of term equalities efficiently [13]. These equalities can be quantified, e.g., theorems and lemmas, that can be instantiated and used arbitrarily many times [15]. Or they can be derived from the context, e.g., pattern matches or program semantics [15]. Or they can be supplied by the user, e.g., code assertions or invariants [3].

Authors' Contact Information: George Zakhour, University of St. Gallen, St. Gallen, Switzerland, george.zakhour@unisg.ch; Jahrim Gabriele Cesario, University of St. Gallen, St. Gallen, Switzerland, jahrimgabriele.cesario@unisg.ch; Pascal Weisenburger, University of St. Gallen, St. Gallen, Switzerland, pascal.weisenburger@unisg.ch; Guido Salvaneschi, University of St. Gallen, St. Gallen, Switzerland, guido.salvaneschi@unisg.ch.

Equality Saturation and E-Graphs. In this work we focus on provers that favor the first technique, dubbed *equality saturation* [20], over those that develop complex heuristics. Precisely those that apply all equalities at once and hence "saturate their knowledge".

There are two data structures to store such a large, often infinite, amount of equalities: the *union-find* [18] and the *e-graph* [12]. Given a set of unquantified equalities, the former is an efficient implementation of the smallest equivalence relation that closes over these equalities. The latter is an extension of the union-find that supports uninterpreted function symbols and efficiently implements the smallest *congruent* equivalence relation, i.e. the relation in which $f(x) = f(y)$ whenever $x = y$.

For quantified equalities, provers employ *e-matching* [7], a procedure that searches an existing set of terms for assignments to quantified variables. These assignments are later used to add new equalities to the e-graph. These new equalities could themselves be reused for other quantified equalities to discover further equalities. Thus, e-matching happens in a loop until no new equalities are discovered, i.e., until the e-graph saturates.

E-Graphs Extensions. With e-graphs, e-matching, and the saturation loop, one can support further theories. For example, to support disequalities, the disequality relation can be *embedded* into the e-graph, i.e., reified into its own uninterpreted function, while saturating the e-graph with respect to the axioms of disequality supplied as quantified equalities [23] To support reasoning modulo alpha-equivalence for binders in lambda expressions or pattern matching, de Bruijn indices/levels can be used with the special *shift* function to shift indices when saturating the e-graph [11]. To support reasoning by cases, one could contextualize statements inside equalities by reasoning inside of a special function symbol *assume* and saturating based on its axioms [6].

However saturating the e-graph on fundamental axioms takes valuable resources which could be better spent in saturating on lemmas and theorems, especially as the saturation loop is the bottleneck of automated provers. Hence, in recent

years, a significant effort has been extending the e-graph to support these theories natively and hence more efficiently. For example disequality edges extend the e-graph to support disequalities natively [23], slotted e-graphs allow reasoning modulo alpha-equivalence [16], and more recently versioned e-graphs [1] allow efficient reasoning by case analysis.

Proofs from E-Graphs. To increase confidence in the result of these provers, they often produce proofs justifying their answer. To do so, the e-graph, which traditionally only answers boolean queries, whether an equality holds or not, is extended to also answer *why* an equality holds.

The proof produced from e-graphs is a sequence of user-supplied equalities, sometimes nested inside congruences, that transform the left-hand-side of a queried equality to the right-hand-side. Thus, by justifying all equalities constitutes a proof that can be checked in a separate independent tool.

These automated provers can also be subsystems of larger provers. For example, the state-of-the-art Lean4 tactics `grind` [21], the e-graph library `egg` [10, 22], and the Rocq Coquetier tactic [5] offload some of the proof to an e-graph-based solver by serializing the environment and then converting the e-graph's proof back into a Lean or Rocq proof.

For the reasons outlined above, proof production is paramount to provers. Alas, proof production for the described state-of-the-art extensions is missing from the literature.

Contributions. In this paper, we describe a first adaptation to the proof production algorithm that supports the extension for proof by cases. We develop a lightweight automated inductive theorem prover Vegie that implements the extension and the proof production. We describe the implementation of Vegie, its automatically generated proofs, and the features required by independent checkers to validate them. In summary, our contributions are:

- A proof production algorithm for versioned e-graph (section 3).
- Vegie, an automated inductive theorem prover based on versioned e-graphs, that can reason by induction, by cases, and reductio ad absurdum (section 4).
- A case study comparing the proofs generated by Vegie to the state-of-the-art Lean `grind` tactic (section 5).

2 Background

E-Graphs have been introduced by Nelson in his seminal PhD thesis as an efficient data structure to store and query a set of equalities [12]. It can be seen as an extension of the union-find data structure [18]. In this section we start by describing union-finds, e-graphs, traditional proof extraction algorithms, and the versioned e-graph extension.

2.1 E-Graphs

Union-Find. The union-find data structure [18] efficiently stores and queries a set of equalities between constants. It

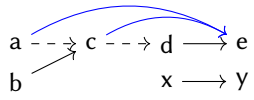

Figure 1. The union-find constructed from (1) $a = c$, (2) $x = y$, (3) $c = b$, (4) $c = e$, (5) $b = e$ (redundant), and (6) $d = e$. After finding the root of a, path compression removes the dashed edges and replaces them with the solid blue ones.

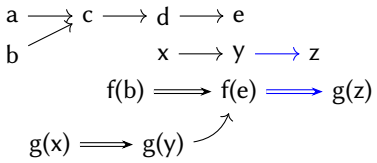

Figure 2. The e-graph starting from the same equalities of Figure 1, adding (7) $f(b) = g(x)$ adds two congruence edges (double black), then adding (8) $y = z$ (blue) adds a further congruence edge (double blue).

constructs a forest where all equal constants belong to the same tree as follows. In the empty union-find, there are as many trees as elements in the universe. To *union* two elements, we *find* the roots of the trees of both elements and we add one as a child to the other. To decide if two elements are equal, i.e., that they belong to the same tree, we traverse the tree of each element upwards towards the root and check if the roots are the same. An important optimization to improve the performance of the union-find is *path compression* [19]. After traversing the tree to find the root of an element, all the nodes along the path are moved to be direct children of the root. Figure 1 illustrates the union-find constructed from user-supplied equalities with path compression.

E-Graph. An e-graph [12] is an extension of the union-find that supports reasoning about uninterpreted functions and stores equalities between function applications (including constant functions). More precisely, it extends the equivalence relation of the union-find to be a congruent relation, i.e., whenever $x = y$ is discovered then $f(x) = f(y)$ is automatically implied for any function symbol f. This is the *congruence invariant* of the e-graph.

In its simplest form, an e-graph is a union-find and a hash-consing table, which maps each function application to a unique identifier, called *e-class*. The internal union-find is then defined over these e-classes and maintains the equivalence relation of the e-graph. Function applications are modelled as *e-nodes*, which are function symbols applied to a list of e-class arguments. Thus, each e-node efficiently represents a set of equivalent function applications, i.e., all the function applications whose arguments belong to the same e-classes as the e-node's arguments.

An e-graph supports the same *find* and *union* operations as the union-find. The difference is that equating two e-classes

maintains the congruence invariant. The invariant is easily restored by equating all e-nodes with their respective *canonicalization*, obtained by replacing each e-class argument with its respective root in the union-find. We call the additional edges resulting from congruence the *congruence edges*. Figure 2 illustrates the e-graph constructed from the same user-supplied equalities as Figure 1 with the additional effect of congruence.

2.2 Proofs from E-Graphs

The Proof Forest. Proof generation for the union-find and the e-graph has been introduced by Nieuwenhuis and Oliveras [14]. A proof that $a = b$, in the union-find, is a sequence of user-supplied equalities that form a path from a to b through intermediary elements. A proof of equality between two elements in the union-find is almost precisely the path connecting both elements in their tree since each edge is the result of a user-supplied equality. What prevents these edges from being directly used is (1) that if $a = b$ is supplied and either is not a root, then the edge is not connecting a to b, and (2) path compression moves the target of the edge to an element potentially different than the one in the user-supplied equality.

To fix this problem, an auxiliary data structure, the *proof forest* is used. Whenever a and b are equated, then a is connected to b by a *proof edge* in the forest, unless a cycle is created. This last criterion guarantees efficient proof generation by avoiding cycle detection.

Proof generation in e-graphs delegates to the underlying union-find, except some subproofs may be due to congruence. Since these are not user-supplied, they must be themselves proven. For example, if $f(x, y) = f(z, w)$ by congruence, then the proofs of $x = z$ and $y = w$ must be produced.

The Proof Graph for Smaller Proofs. A criticism of the algorithm constructing the proof forest is that it may ignore some user-supplied equalities which would lead to shorter proofs. Flatt et al. [10] describe a case where a circuit optimizer finished in under a minute, yet the corresponding equivalence proof was so large that its verification required four hours. They propose an algorithm for shorter proofs by relaxing the acyclicity of the proof forest, allowing it to be a *proof graph*. Thus, a short proof is a short path in the graph.

While this approach is straightforward for the union-find, the challenge for e-graphs stems from congruence edges, which expand to more proof obligations whose "cost" is not obvious during traversal of the proof graph. As a result, Flatt et al. [10] approximate the cost of congruence edges.

Figure 3 shows the proof graph underlying the e-graph from Figure 2. The proof that $f(e) = g(y)$ is the path $(c_0)^{-1}$, (7), (c_1). First, it follows a congruence edge for f backwards, requiring a subproof that $b = e$, which is the path consisting of the single step (5), or alternatively the path $(3)^{-1}$, (4). Second, it follows the user-supplied equality (7), i.e., $f(b) =$

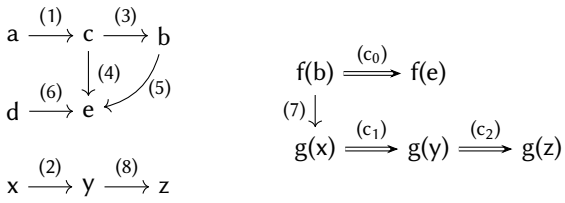

Figure 3. Proof graph from all equalities of Figures 1 and 2.

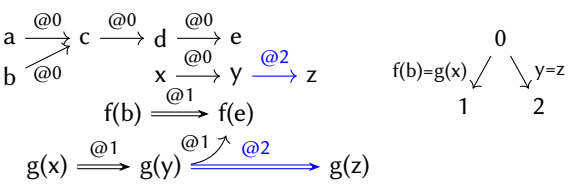

Figure 4. The versioned e-graph with the equalities of Figure 1 at the root version @0 (left), and its version tree (right). Two cases are created: (7) $f(b) = g(x)$ at version 1; and (8) $y = z$ at version 2. Notably, $g(x) = g(z)$ never holds, while $x = z$ holds at version 2 but not 1.

$g(x)$. Lastly, a congruence edge for g is expanded to the subproof of $x = y$, which is the path the single step (2). Thus, the full proof is: $\text{congruence}_f((5))^{-1}$, (7) $\text{congruence}_g((2))$.

2.3 Versioned E-Graphs

Traditional e-graphs encode a single global equality relation, which limits their effectiveness in branching contexts such as proof by case analysis. In automated theorem proving, proofs are often organized across cases, such that each proof branch evolves into a different set of valid equalities. These contextual equalities are then handled by maintaining multiple e-graphs, e.g., one for each branch in the proof. Recently, *versioned e-graphs* [1] have been proposed as a more efficient solution, which maximizes sharing between multiple equivalence relations in a single e-graph.

The key idea of versioned e-graphs is to maintain a non-empty *version tree*, where each node uniquely identifies one equivalence relation, or *version*. Thus, versions are organized in a hierarchy of increasingly refined equivalence relations, in which the *root version* is the coarsest equivalence relation.

To maximize sharing, versioned e-graphs maintain a single hash-consing table and a *versioned union-find*, whose edges are annotated with the version, in which the equality holds. The *find* and *union* operations have are adapted to versioning: When searching the root of an element at a version, the operations are only allowed to follow edges at the version or at its ancestors; when equating two elements at a version, their equality is propagated to all descendant versions.

3 Proof Extraction for Versioned E-Graphs and Algebraic Datatypes

3.1 Theory

To motivate the need for case analysis, we develop the proof production algorithm in the theory of data types. This theory is parametrized by a set of types τ, a set of constructors κ, and an association $\Gamma : \tau \to \mathcal{P}(\kappa)$ that maps a set of constructors to a type. The expressions e in this theory are constants, applications to uninterpreted functions, and applications to constructors. These four axioms are assumed:

1. If $\tau_1 \neq \tau_2$ then $\Gamma(\tau_1) \cap \Gamma(\tau_2) = \emptyset$,
2. If $e : \tau$ then $e = k(\overline{e'})$ for some $k \in \kappa$ and $\overline{e'}$,
3. if $k(\overline{e_1}) = k(\overline{e_2})$ then $\overline{e_1 = e_2}$ for some $k \in \kappa$, and,
4. if $k_1 \neq k_2 \in \kappa$ then $k_1(\overline{e_1}) \neq k_2(\overline{e_2})$ for all $\overline{e_1}$ and $\overline{e_2}$.

Without the ability to reason by cases it is nigh impossible to make use of the second axiom. And without reductio ad absurdum "from false follows anything", the last axiom is also impossible to make use of. This motivates the following syntactic definition of proofs in Definition 3.1.

A *proof* that $e_1 = e_2$ is either (1) by reflexivity when e_1 is identical to e_2, (2) by definition when e_1 is defined to be e_2 in the environment, (3) by rewriting the top-level expression with some quantified rule r specialized to some expressions $\overline{e}$, i.e. that $r(\overline{e})$ is $e_1 = e_2$, (4) by congruence of some function f, i.e. $e_1 = f(\overline{e'_1})$ and $e_2 = f(\overline{e'_2})$ with $\overline{p}$ being the proofs of $\overline{e'_1 = e'_2}$, (5) by sequencing two or more proofs, (6) by injectivity of some constructor κ, i.e. e_1 and e_2 are the n-th argument to some equal construction because of p, (7) by reductio ad absurdum, i.e. because two different constructions can be proven equal by p, and finally (8) by case analysis.

Definition 3.1 (Proof Syntax)**.** A proof p has the shape:

$$
\begin{aligned}
p ::= {} & \texttt{refl} \mid \texttt{definition} \mid \texttt{rewrite}\ r\ \overline{e} \\
& \mid \texttt{congr}\ f\ \overline{p} \mid \texttt{seq}\ \overline{p} \mid \texttt{injective}\ n\ (\kappa\ \overline{e_1})\ (\kappa\ \overline{e_2})\ p \\
& \mid \texttt{absurd}\ (\kappa_1\ \overline{e_1})\ (\kappa_2\ \overline{e_1})\ p \mid \texttt{cases}\ \overline{h : e_1 = e_2 \Rightarrow p}
\end{aligned}
$$

3.2 In the Versioned E-Graph

Insight. The main insight of versioned e-graphs is that all edges in the union-find and in the e-graph must be labeled by versions. This is no exception for the proof graph. We propose that a proof is also versioned, i.e., that just as one asks whether two expressions are equal *at a version*, one would also ask *why* they are equal at a version.

Sketch of Algorithm. By labeling the edges of the proof graph with versions, it can be traversed in the same fashion as the versioned union-find. In other words, a proof that e_1 is equal to e_2 at some version v consists of a path of edges labeled (l, v') with v' being an ancestor version of v.

Unfortunately, this alone is not enough, as it does not make use of the theory's second axiom: that the constructors are exhaustive for a type, enabling case analysis. To exploit this completeness in proof production, we allow traversing paths also in the subversions where a proof is valid only if there is a path in every subversion. Therefore, proofs in the versioned e-graph are no longer paths, but trees instead.

Moreover, when searching for a proof, we cannot search too deep in the version tree or the proof grows too large. Thus, the proof search in the proof graph must avoid subversions, unless strictly necessary. To achieve this, we add an additional cost to proof edges that grows exponentially with the difference in versions, i.e., an edge one version deep costs w, while one version two deep costs w^2, and so on.

Proof Edge Labels. Proof edges for case analysis are not realistic due to the aforementioned proof explosion. Thus, `cases`, much like `rfl` and `seq`, cannot exist directly as labels in the proof graph and are constructed by the algorithm.

More formally, the edges of the proof graph are labeled by (v, l) where v is a version and l as in Definition 3.2

A *proof edge label* denotes equality either (1) by definition, (2) by an application of a rule, (3) by congruence, (4) by injectivity from the equality of two constructions, or (5) by absurdity from the equality of two constructions.

Definition 3.2 (Proof Edge Labels)**.** A label l is:

$$
\begin{aligned}
l ::= {} & \texttt{definition} \mid \texttt{rewrite}\ r\ \overline{e} \mid \texttt{congr} \\
& \mid \texttt{injective}\ n\ e_1\ e_2 \mid \texttt{absurd}\ e_1\ e_2
\end{aligned}
$$

3.3 Algorithm

The proof production algorithm is in two steps. First, *traversal* finds a path from some expression `a` to another expression `b` in a version `v` in the proof graph, using a custom *cost function*. Then, *conversion* produces a proof from the path.

We can express this in the following pseudo-code:

```
def proof(a, b, v):
  path = shortest_path(proof_graph, a, b, cost(v))
  if length(path) == 0: return Proof.refl
  return Proof.seq(path.map(convert))
```

The cost of an edge at version `v` between two expressions `a` and `b` annotated with version `ev` and `label` is:

```
def cost(v, a, b, label, ev):
  if ev.is_ancestor_of(v) or ev == v or (v .. ev).all_are_disj():
    return estimate(version, a, b, label, ev)
  else if ev.is_descendant_of(v):
    for v' in v .. ev:
      if not equal(src, trgt, v'):
        return Inf
    return size(proof_graph) ^ distance(v, ev)
  else:
    return Inf
```

Three cases are of interest. The last, if the queried version is unrelated to the annotated one, then the edge cannot be traversed, thus it has infinite cost. The second, if the annotated version is a descendant of the current one, then the equality must hold at every version in between. The cost of that edge is exponential in the distance between the versions to avoid traversing subversions unless required. Finally, if the annotated version is equal to or ancestor of the queried version, then the edge should be estimated as is done in [10].

Note that the edge is valid also when all the versions in between are disjunctions, i.e., only one case has to be proven. Disjunctions are used for exploration, e.g., the prover may fork its current version as a disjunction on case expressions, and explore each case expression in its own branch.

The conversion from an edge between two expressions `a` and `b` annotated with a version `ev` and a `label` into a proof is:

```
def convert(a, b, label, ev):
  if ev.is_ancestor_of(v) or ev == v or (v .. ev).all_are_disj():
  match label:
    Label.definition => Proof.definition
    Label.rewrite(r, args) => Proof.rewrite(r, args)
    Label.injective(n, a, b) =>
      Proof.injective(n, a, b, proof(egraph, a, b, v))
    Label.absurd(a, b) =>
      Proof.absurd(a, b, proof(egraph, a, b, v))
    Label.congr =>
      Proof.congr(a.func, a.args.zip(b.args).map(proof(_, _, v)))
  else:
    return cases(v.subversions().map(v' =>
      (v'.condition, proof(a, b, v'))))
```

The transformation is trivial for proofs by definition, by rewrites, and classic by congruence, where proofs of the pairwise equality of the arguments are produced. For injectivity and reductio ad absurdum, a proof for the two constructions is generated. For case analysis, we associate each proof at every subversion with the condition for the fork.

4 Vegie

To evaluate the proof production capabilities of e-graphs across proof branches, we implemented Vegie, an automated inductive theorem prover on top of versioned e-graphs [1].

Language Features. Vegie's specification language is a functional first-order language with immutable identifiers. Much like the SMTLIB format [2], our language is command-oriented and its syntax is based on S-expressions. The available commands are: (1) datatype declarations, (2) function signature declarations, (3) function definition, (4) expression evaluation, and (5) quantified theorem statements. All theorems are stated as equations of expressions. For example, the left-distributivity of multiplication over addition is:

```
(theorem mult_dist_add ((?x nat) (?y nat) (?z nat))
  (mult ?x (add ?y ?z))
  (add (mult ?x ?y) (mult ?x ?z)))
```

Here, `mult_dist_add` is the name of the theorem; the list `((?x nat) (?y nat) (?z nat))` are the quantified variables, whose names start with the sigil `?`, followed by their type; the second line is the left-hand-side of the equation; and the third line is the right-hand-side. Both sides must be proven equivalent for the theorem to be true.

Algebraic Datatypes. Vegie does not come with any built-in primitive types. Users are expected to define their own, using inductive algebraic data types.

For example, booleans can be defined as `(datatype bool true false)`, stating that the datatype `bool` has two nullary constructors `true` and `false`. The Peano natural numbers can be defined as `(datatype nat zero (succ nat))`, stating that the type `nat` has a nullary constructor `zero` and a unary constructor `succ` accepting a `nat`. Arbitrary-length bit-vectors may be defined as `(datatype bitvec (bit bool)(vec bitvec bool))`.

To reason about algebraic datatypes we endow Vegie with two axioms: (1) that constructors are injective functions, i.e., if `(vec v0 b0)` and `(vec v1 b1)` are known to be equal, then `v0` is equal to `v1` and `b0` is equal to `b1`, and (2) that values constructed from different constructors are unequal, i.e., that `(succ n)` is always unequal to `zero`.

This last axiom is used to reason by *reductio ad absurdum*. If in a specific context, we discover that `(succ n)` and `zero` are equal, then we can immediately prove the theorem.

Pattern Matching. Vegie supports *pattern matching* to make use of algebraic datatypes. Below follows an example:

```
1 (type add (nat nat) nat)
2 (def add (?x ?y)
3   (case ?x
4     (zero ?y)
5     ((succ ?x) (succ (add ?x ?y)))))
```

In line 1, we declare the `add` function, which takes two `nat` and returns a `nat`. In line 2, we define `add`, binding the function inputs to variables `?x` and `?y`. Line 3 contains the case expression examining `?x`. Lines 4 and 5 are the pairs of pattern–expression that define the case expression.

We enforce that patterns do not overlap.

Non-overlapping patterns may be cumbersome in certain situations, however, they allow us to reason about each case independently, without carrying around the negation of all previous patterns. For example, consider this definition of less-than-or-equal `leq` of a pair of natural numbers:

```
(datatype pair (pair nat nat))
(type leq (pair) bool)
(def leq (?p) (case ?p
  ((pair zero ?x) true)
  ((pair ?x zero) false)
  ((pair (succ ?x) (succ ?y)) (leq ?x ?y))))
```

Without a careful treatment of case expressions it is possible to derive that `true` and `false` are equal: `(leq zero zero)` reduces to `true` by the first pattern and to `false` by the second pattern. Therefore, Vegie rejects the previous definition on grounds that the patterns overlap. Note that a more convenient pattern matching construct for a surface language could be easily compiled to non-overlapping patterns.

We explore each branch of the pattern match in a different e-graph version. Further, using the disequality edge extension of e-graphs [23], Vegie easily keeps track of all the disequalities between the case expression's scrutinee and the patterns of the branches explored in another version.

Recursion and Induction. As shown in the previous examples, Vegie supports recursion. Although Vegie does not support forward referencing, mutual recursion can be achieved by separating function declarations and definitions.

For example, we can define the `even` and `odd` predicates over Peano numbers mutually as follows:

```
(type even (nat) bool)
(type odd (nat) bool)

(def even (?x) (case ?x
  (zero true)
  ((succ ?x) (not (odd ?x)))))

(def odd (?x) (case ?x
  (zero false)
  ((succ ?x) (not (even ?x)))))
```

Later, we can state theorems such as:

```
(theorem even_plus_2 ((?x nat))
  (even (succ (succ ?x)))
  (even ?x))
```

Recursion unlocks proof by induction. For example, consider the theorem that `add` is commutative:

```
(theorem add_comm ((?x nat) (?y nat)) (add ?x ?y) (add ?y ?x))
```

The proof in Vegie unfolds `add` and performs case analysis on `?x`. In the branch where `?x` is `(succ ?y)`, we can apply `add _comm` – effectively the induction hypothesis – as a theorem to the recursive call since `?y` is structurally smaller than `?x`.

The current implementation of Vegie checks that structural lexicographic ordering of arguments is decreasing before applying the inductive hypothesis, which ensures the application of the inductive hypothesis is sound. Without it, one could derive an unsoundness, that `true` is equal to `false` using the definition: `(def weird ()(not weird))`. To guarantee soundness, we further need to ensure that all functions terminate, for which we also check that the lexicographic ordering of arguments structurally decreases for every recursive call.

The Proof-Search Loop. Vegie's main search loop contains three components. (1) First, it applies the following rules in a loop: (a) the non-overlapping axiom of constructors, (b) the injectivity of constructors, and (c) the inductive hypothesis and previously proven theorems. These rules are applied for a few iterations before carrying through the main proof search loop. (2) Then, Vegie finds an appropriate case expression to do case analysis on. (3) Finally, it finds appropriate definitions to expand.

5 Case Study

To get an impression of the quality of the proofs produced by Vegie, we compare the generated proof of the commutativity of Peano number addition to the one generated by the state-of-the-art `grind` tactic in Lean4.

After defining the Peano numbers and `add` as usual, we supply four (proven) lemmas to Vegie and `grind`: `zero` is a left and right unit of `add` and if one of the arguments of `add` is `succ x`, then `succ` can be applied outside.

Since `grind` cannot do induction automatically, it fails to find a proof. After manually introducing the induction hypothesis, `grind` produced a proof by contradiction which requires classical reasoning. This is expected as `grind`'s internals are SMT-like. The total size of the proof is 1.1 KB for the base case and 1.6 KB for the inductive case.

The proof generated by Vegie is 395 B and is the following:

```
1  (cases
2    (_h0 (?y zero)
3      (seq
4        (congr add refl (apply _h0))
5        (apply add_zero_right ?x)
6        (apply <- add_zero_left ?x)
7        (congr add (apply <- _h0) refl)))
8    (_h1 (?y (succ ?z))
9      (seq
10       (congr add refl (apply _h1))
11       (apply add_succ_right ?x ?z)
12       (congr succ (apply add_comm ?x ?z))
13       (apply <- add_succ_left ?z ?x)
14       (congr add (apply <- _h1) refl))))
```

It performs case analysis on one of the quantified variables using the `cases` tactic which has two branches: one that binds `_h0` to the equality `?y = zero` and one that binds `_h1` to `?y = (succ ?z)`. Each case applies a sequence of congruences (left-to-right and right-to-left). Most importantly, it's clear that the proof is by induction since on line 12 the `add_comm`, the very theorem we're trying to prove, is used.

6 Related Work

E-graphs originate from the congruence-closure algorithm by Nelson [12], which maintains equivalence classes of terms under congruence efficiently.

Equality saturation was later introduced as a compiler optimization technique [20], which represents many equivalent program expressions simultaneously and applies rewrites until saturation. The *egg* library improves the scalability of equality saturation with rebuilding and efficient rule scheduling [22]. The egg framework also introduced *e-class analyses* which associate abstract information with equivalence classes and maintain it incrementally as the e-graph evolves [22], enabling optimizations such as constant propagation.

E-matching provides efficient rewrites based on pattern matching modulo equivalence. E-matching techniques were developed in the context of SMT solvers, where they are used to instantiate quantified formulas over congruence structures [7, 9]. These techniques are implemented in systems such as Z3 [8] and have been adapted for efficient rewrite matching in equality saturation engines such as egg [22].

Many e-graph implementations represent equivalence relations but do not directly encode disequality constraints. Instead, disequalities are typically handled externally or encoded indirectly using auxiliary terms or solver-side constraints. Recent work [23] proposes *dis/equality graphs*, an extension of e-graphs that represents disequality relations explicitly alongside equality edges.

To increase confidence in e-graph-based reasoning, researchers have explored proof generation objects from reasoning based on congruence closure. Early work introduced

proof-producing congruence closure, extending union–find-based algorithms with an *explain* operation that reconstructs a derivation showing why two terms are equal [14]. More recent work explores producing compact certificates for equality reasoning [10] for generating small congruence-closure proofs with greedy methods.

7 Conclusion & Future Work

We have presented a proof extraction algorithm for versioned e-graphs in the setting of algebraic datatypes. Our approach extends traditional e-graph proof production with proof primitives for injectivity, contradictions between constructors, induction, and case analysis, allowing proofs to be extracted from branching reasoning contexts rather than only from a single global equivalence relation. We implemented these ideas in Vegie, an automated inductive theorem prover based on versioned e-graphs, and used it to study the shape and size of the resulting proof objects, including an initial comparison with Lean's grind tactic.

In the future we wish to support disequality edges and slotted e-graphs in the proof extraction as well as have Vegie produce proofs for LiquidHaskell, Lean4, and Rocq.

References

[1] Anonymous. 2026. Versioned E-Graphs. *Proceedings of the ACM on Programming Languages* 10, PLDI, Article 118 (Oct. 2026), 29 pages.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7*. SMT-LIB. https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-02-05.pdf. Accessed 2026-03-14.

[3] Nikolaj S Bjørner, Kenneth L McMillan, and Andrey Rybalchenko. 2012. Program Verification as Satisfiability Modulo Theories. *SMT@ IJCAR* 20 (2012), 3–11.

[4] Lasse Blaauwbroek, David M. Cerna, Thibault Gauthier, Jan Jakubův, Cezary Kaliszyk, Martin Suda, and Josef Urban. 2024. *Learning Guided Automated Reasoning: A Brief Survey*. Springer Nature Switzerland, Cham, 54–83. https://doi.org/10.1007/978-3-031-61716-4_4

[5] Thomas Bourgeat. 2023. *Specification and Verification of Sequential Machines in Rule-Based Hardware Languages*. PhD thesis. Massachusetts Institute of Technology. ProQuest Dissertations & Theses, document ID 30672271.

[6] Samuel Coward, Theo Drane, and George A. Constantinides. 2024. Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 14 pages. https://doi.org/10.1109/TCAD.2024.3483096

[7] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[9] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473.

[10] Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small proofs from congruence closure. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 75–83.

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

[12] Charles Gregory Nelson. 1980. *Techniques for program verification*. Stanford University.

[13] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

[14] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*. Springer, 453–468.

[15] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98.

[16] Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1888–1910.

[17] Stephan Schulz and Martin Möhrmann. 2016. Performance of Clause Selection Heuristics for Saturation-Based Theorem Proving. In *Automated Reasoning*, Nicola Olivetti and Ashish Tiwari (Eds.). Springer International Publishing, Cham, 330–345.

[18] Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

[19] Robert E. Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (March 1984), 245–281. https://doi.org/10.1145/62.2160

[20] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[21] The Lean Project. 2025. *The Lean Language Reference*. Lean FRO. https://lean-lang.org/doc/reference/4.29.0-rc6/. Section 16, "The grind tactic", version 4.29.0-rc6, accessed 2026-03-14.

[22] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[23] George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2282–2305.