Disequalities in E-Graphs: An Experiment

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

This talk explores the integration of disequalities into e-graphs for enhancing the efficiency of automated theorem provers. We discuss two existing approaches, which we implement in the egg e-graph library, presenting preliminary results on comparing their effectiveness. Our initial experiments demonstrate the feasibility of integrating disequalities into e-graphs implemented in egg, with promising results suggesting improved efficiency in the proof search algorithm. We plan to refine this approach, integrate it into our Propel automated theorem prover, and make our extensions to egg available to the research community.

1 Introduction

In this talk, we present our ongoing work of integrating disequalities into e-graphs. We present the two prevalent approaches to support disequalities and implement both in the egg e-graph library [10], discuss them, and provide an initial comparison.

Our prior research on the Propel automated prover [11, 12] suggests that reasoning about disequalities is beneficial in theorem provers. While some SMT solvers like Z3 [2] and OpenSMT [1] integrate disequalities into their e-graphs, it is less common for automated theorem provers to do so, with Simplify [3] being one of the few exceptions. We believe that integrating an e-graph as an efficient data structure to store equivalent terms in Propel would improve it performance. Yet, this approach requires the integration of disequalities resoning into e-graphs – presently, Propel does not use e-graphs to determine the equality or disequality of terms but relies on an ad-hoc mechanism. Sections 2 and 3 describe two alternative solutions to integrate disequalities into e-graphs.

Equality Rewriting in Propel. Propel [11, 12] is an automated theorem prover targeting fundamental algebraic properties such as commutativity, associativity, idempotency, reflexivity, transitivity, symmetry, and others. Propel's core feature is the ability to reason about such properties by (a) applying rewrite rules derived from the properties and (b) deducing the equality or disequality of terms. Focusing on algebraic properties enables Propel to successfully discovering a proof for cases where other state-of-the-art automated theorem provers and SMT solvers fail.

To prove, for example, that multiplication is commutative $x \times y = y \times x$, Propel applies a sequence of rewrite rules to the terms on the left-hand and right-hand side of the equation, trying to derive an equation that is trivially true by syntactic equality $e = e$. Rewrite rules are derived from the algebraic properties of the involved functions. For example, if multiplication $\times$ is defined in terms of addition $+$ and the prover successfully verified that addition is commutative, it can use the rewrite $x + y \rightsquigarrow y + x$ when reasoning about the definition of $\times$.

Internally, Propel maintains a collection of terms that were discovered to be equal by the prover. In the current implementation, equal terms are stored as entries within a map, which models the rewrite rules from key terms to value terms that is assigned to the respective key. The prover explores various rewrite possibilities – prioritizing the most "promising" terms according to a heuristics – until both sides of the equation are rewritten into identical terms.

E-Graphs With Disequalities For Propel. Instead of the approach based on a map above, we consider leveraging e-graphs for reasoning about the equality of terms. Yet, in Propel, it essential to reason not only about known equalities of terms, but also about known disequalities.

Knowing that terms are unequal allows for discharging prove obligations early, thus pruning the search space substantially. Some algebraic properties, like the antisymmetry of relations, crucially rely on the disequality of terms, e.g., a relation R is antisymmetric if and only if for two terms x and y with $x \neq y$ holds that $xRy \rightarrow \neg yRx$ or – equivalently – if a relation R is known to be antisymmetric and xRy then either $x = y$ or $x \neq y \wedge \neg yRx$.

In such cases, reasoning about disequalities is important. Yet, popular high-performance implementations of e-graphs, such as egg [10] do not inherently support them. Next, we consider two methods to support disequalities.

2 Method 1: Equality Embedding

This method adds support for disequalities to e-graphs by reifying the equality relation constructed by the e-graph explicitly into the language whose terms constitute the nodes of the e-graph. Thus, this embedding ensures the invariant that two terms `t1` and `t2` are equal if and only if the terms `(eq t1 t2)` and `true` are equal. In the rest, we use the syntax defined by egg's `SymbolLang` [5] for terms, and its corresponding pattern syntax [4] for rules.

This embedding is constructed by first introducing a special symbol, eq, to represent equality, and second saturating the e-graph with the axioms of equality: (1) reflexivity: `(eq ?x ?x)` rewrites to `true`, (2) symmetry: `(eq ?x ?y)` rewrites to `(eq ?y ?x)`, and (3) transitivity: `?a = true = (eq ?x ?y)` and `?b = true = (eq ?y ?z)` rewrites to `?a = ?b = (eq ?x ?z)`. Encoding the knowledge that `t1` and `t2` are unequal is thus as simple as equating `(eq t1 t2)` to `false` in the e-graph.

Given this encoding, two terms `t1` and `t2` are known to be equal if `(eq t1 t2)` equals `true` and are known to be unequal if `(eq t1 t2)` equals `false`. Checking whether the given equalities and disequalities are inconsistent, i.e., that some contradiction exists, is as simple as checking whether `true` and `false` are equal.

2.1 Equality Embedding with Egg

Listing 1 shows a self-contained example using the egg e-graph library to embed some equalities and disequalities. After adding all the provided equalities, the program checks whether a contradiction can be found.

Listing 1. Equality Embedding in egg.

```
1  use egg::{*, rewrite as rw, multi_rewrite as mrw};
2
3  macro_rules! parse {($e:expr) => { $e.parse().unwrap() }}
4
5  fn main() -> () {
6    let mut g: EGraph<SymbolLang, ()> = Default::default();
7
8    [ ("x", "(f (f x))")
9    , ("x", "(f (f (f x)))")
10   , ("(eq (f x) x)", "false")
11   ].iter().for_each(|(lhs, rhs)| {
12     let id1 = g.add_expr(&parse!(lhs));
13     let id2 = g.add_expr(&parse!(rhs));
14     g.union(id1, id2);
15   });
16
17   let mut runner = Runner::default().with_egraph(g).run(&[
18      rw!("e1"; "(eq ?x ?x)" => "true"),
19      rw!("e2"; "(eq ?x ?y)" => "(eq ?y ?x)"),
20     mrw!("e3"; "?a = true = (eq ?x ?y),
21                 ?b = true = (eq ?y ?z)"
22             => "?a = ?b = (eq ?x ?z)"),
23   ]);
24
25   let g = &mut runner.egraph;
26   let t = g.add_expr(&parse!("true"));
27   let f = g.add_expr(&parse!("false"));
28
29   let c = if g.find(t) == g.find(f) { "Y" } else { "N" };
30   println!("Contradiction: {c}");
31 }
```

The loop on Line 11 traverses the list of pairs on Lines 8 to 10 containing the left-hand-side and the right-hand-side of some (dis)equality. The added (dis)equalities are $x = f(f(x))$, $x = f(f(f(x)))$, and $f(x) \neq x$. The runner on Line 17 saturates the e-graph with the axioms of equalities in Lines 18 to 22. Lines 26 and 27 looks up the e-classes of the `true` and `false` terms, respectively, and Line 30 prints `Contradiction: Y` if `true` and `false` are equal; `Contradiction: N` otherwise.

Since the provided set of equalities and disequalities are not consistent the output of the program is, as expected, `Contradiction: Y`, i.e., a contradiction exists. Changing the disequality on Line 10 into an equality makes the equations consistent and the output is `Contradiction: N`, i.e., no contradiction exists. Similarly, the equality on Line 8 may also be changed into a disequality to eliminate the contradiction.

2.2 The method in the Wild

Z3 [2] utilizes an e-graph to store equalities when solving in the Uninterpreted Functions (UF) theory. To support disequalities, a common input for SMT solvers, Z3 resorts to an embedding technique that is in essence the one above.

Listing 2 shows an excerpt from Z3's e-graph implementation of a method that checks if two given terms are unequal.[1]

Listing 2. Excerpt of Z3's check for disequality.

```
1 bool egraph::are_diseq(enode* a, enode* b) {
2     enode* ra = a->get_root(), * rb = b -> get_root();
3     // ...
4     enode* r = tmp_eq(ra, rb);
5     if (r && r->get_root()->value == l_false)
6         return true;
7     return false;
8 }
```

On Line 2, Z3 uses its union-find data structure [6] to compute the e-classes of the given e-nodes. On Line 4, it searches for the e-class of a language-level equality expression with the input's e-classes on either side. On Line 5, it checks whether that e-class is the same as the e-class containing falsum whose canonical representative is `l_false`.

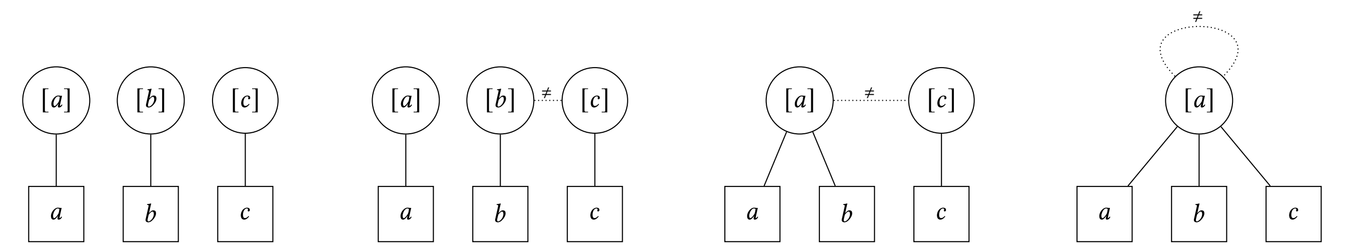
3 Method 2: Disequality Edges

E-graphs were introduced by Nelson in his PhD thesis on program verification [7]. As disequalities are common in statements pertaining to program verifications it is no surprise that Nelson designed e-graphs in a way that disequalities be easily represented.

E-graphs – ignoring disequalities – have two kinds of edges: (1) node–class edges encoding the fact that a node belongs to some class (equivalence), and (2) class–node edges encoding that a hole in the node can be occupied by any member of the class (congruence). To support disequalities additional structure was added to the e-graph: class–class edges that encode that the two classes may not be merged, i.e., that their members are unequal.

Figure 1 shows an example of how a contradiction can be found. Section 3.1 describes through code definitions how the e-graph use disequality edges, i.e., forbid lists, by extending egg.

[1] https://github.com/Z3Prover/z3/blob/9425c4/src/ast/euf/euf_egraph.cpp#L685-L697



(a) An e-graph with three nodes (square): a, b, c each in their own class (circle): $[a]$, $[b]$, $[c]$ respectively.

(b) Adding $b \neq c$ connects the two classes $[b]$ and $[c]$ with a disequality edge (dotted with ≠ above).

(c) Adding $a = b$ and merging $[b]$ into $[a]$ the disequality edge between $[b]$ and $[c]$ becomes between $[a]$ and $[c]$.

(d) Adding $a = c$ and merging $[c]$ into $[a]$ the disequality edge becomes a self-loop indicating a contradiction.

Figure 1. Disequality edges help in finding the contradiction in $b \neq c$, $a = b$, and $a = c$.

3.1 Adding Support for Disequality Edges in egg

We describe the minimal required additions for introducing support for disequality edges in egg.

Adding the* `forbid` *List. Our implementation follows Nelson's encoding of disequality edges as forbid lists in e-classes [7]. Thus, the first addition is to be done on the `EClass` struct[2] by adding a `forbid` field as shown in Listing 3.

Listing 3. Adding the `forbid` field to `EClass`.

```
pub struct EClass<L, D> {
    // ...
    /// The eclasses known to be unequal to this one.
    pub forbid: Vec<Id>,
    // ...
}
```

The `forbid` field is implemented as a list of e-class identifiers `Id`.

Moving* `forbid` *Lists After Unioning. When equating two terms, the e-class of one is removed and all its children are added to the e-class of the other. We must also treat the forbid lists similarly by appending the forbid list of the removed one to the forbid list of the other. In egg, the function responsible for performing the union is the `perform_union` method of the `EGraph` structure.[3] Line 7 of Listing 4 is the necessary addition to that function.

Listing 4. Updating the `forbid` list.

```
1 fn perform_union(
2     // ...
3 ) -> bool {
4     // ...
5     concat_vecs(&mut class1.nodes, class2.nodes);
6     concat_vecs(&mut class1.parents, class2.parents);
7     concat_vecs(&mut class1.forbid, class2.forbid);
8     // ...
9 }
```

Updating all the `class2` references in its forbid list to `class1` is not necessary since egg keeps all the identifiers of all classes, removed or not, in the union-find data structure.

[2]https://github.com/egraphs-good/egg/blob/3231b86/src/eclass.rs
[3]https://github.com/egraphs-good/egg/blob/3231b86/src/egraph.rs

Adding Disequalities. To add a disequality to the e-graph, we define a new function `disunion` to be implemented for the `EGraph` structure. The function's definition is in Listing 5.

Listing 5. Adding a disequality.

```
1 pub fn disunion(&mut self, id1: Id, id2: Id) {
2     let id1 = self.find(id1);
3     let id2 = self.find(id2);
4     self.classes.get_mut(&id1).unwrap().forbid.push(id2);
5     self.classes.get_mut(&id2).unwrap().forbid.push(id1);
6 }
```

On Lines 2 and 3 we find the identifier of the e-classes of the arguments. On Lines 4 and 5 we push to the `forbid` list of each e-class – which must exist by the post-condition of `find` – the identifier of the other.

Checking Disequalities. Listing 6 defines the function `are_unequal` for `EGraph` which can be used to check if two terms are known to be unequal.

Listing 6. Checking a disequality.

```
1  pub fn are_unequal(&self, id1: Id, id2: Id) -> bool {
2      let mut id1 = self.find(id1);
3      let mut id2 = self.find(id2);
4
5      if self.classes[&id1].forbid.len() >
6         self.classes[&id2].forbid.len() {
7           std::mem::swap(&mut id1, &mut id2);
8      }
9
10     self.classes[&id1].forbid.iter().any(|id| {
11         self.find(*id) == id2
12     })
13 }
```

On Lines 2 and 3, we find the e-class identifiers of both arguments. On Line 7, we swap the identifiers if the forbid list of the first is longer than the second, ensuring that we search over the smaller list and making searching asymptotically faster [7]. Finally, on Line 10, we check whether any of the e-class identifiers in the first's forbid list is equal to the second's identifier. Recall that when unioning two e-classes we never update the identifier of the removed e-class in the forbid lists of the other e-classes. To this end, we always find the root identifier in the union-find data structure of every identifier in the forbid list before comparing. This check is known to be constant amortized time [9] and does not introduce any runtime overhead.

Checking Consistency. Finally, in Listing 7, we define the `is_consistent` for `EGraph` which can be used to check that no contradiction exists.

Listing 7. Consistency checking.

```
pub fn is_consistent(&self) -> bool {
    self.classes.values().all(|eclass| {
        let eclass_id = self.find(eclass.id);
        eclass.forbid.iter().all(|id| {
            self.find(*id) != eclass_id;
        })
    })
}
```

The `is_consistent` function simply checks that the root e-class identifiers of all the known e-classes – removed and not – are different from all the root e-class identifiers of the e-classes in their forbid list.

3.2 In the Wild

OpenSMT [1], like Z3 [2], uses an e-graph to store equalities when solving in the UF theory. Unlike Z3, it also stores disequalities using disequality edges as described in Nelson's PhD thesis with the help of a forbid list. Listing 8 shows an excerpt from OpenSMT's e-graph implementation where disequalities are inserted and processed by the e-graph.[4]

Listing 8. Excerpt of OpenSMT's disequality handling.

```
1  bool Egraph::assertNEq ( Enode * x, Enode * y, Enode * r )
2  {
3    Enode * p = x->getRoot( );
4    Enode * q = y->getRoot( );
5    // ...
6    Elist * pdist = new Elist;
7    pdist->e = p;
8    // ...
9    if ( q->getForbid( ) == NULL )
10   // ...
11     q->setForbid( pdist );
12   else
13   // ...
14     q->getForbid( )->link = pdist;
15   // ...
16 }
```

The disequality handler takes two e-nodes `x` and `y` to assert as unequal with the e-node `r` being the reason, i.e. the justification or explanation [8], on Line 2. The last argument is useful when explanations are generated. But for the sake of explaining the excerpt it can be ignored. In Lines 3 and 4, the e-class of each argument is computed using the union-find data structure. To add `x` into the forbid list of `y`, an empty linked list is created on Line 6 whose head is assigned on Line 7 the e-class of `x`. In Line 9, the forbid list of the e-class of `y` is checked for emptiness. If it is empty, it is set to the list created earlier using the `setForbid` method. If it is not empty, it is appended to the existing forbid list on Line 14. The logic to add `y` into the forbid list of `x` is identical and hidden under Line 15.

[4]https://github.com/formalmethods/opensmt/blob/c7f18acd/src/egraph/EgraphSolver.C#L923-L996

4 Evaluation

To evaluate the two methods described in Sections 2 and 3, we conducted the following two experiments.

Setting. We generated 30 K pairs of expressions in egg's SymbolLang language [5] with maximum depth 5. The expressions range over the numbers 1 to 5 (inclusive) and over applications of three functions, `f`, `g`, and `h` of arity 1, 2, and 3, respectively. One example of such a pair is the following: `(f (f (h 4 2 1)))` and `(f (h (g 5 1)(f (h 2 4 3))(g 5 5)))`.

We fix a ratio between the pairs int the 30 K to be treated as disequalities and those to be treated as equalities. As SymbolLang handles the numerals 1, 2, 3, 4, and 5 as symbols and not integers – i.e., they are not inherently different for SymbolLang – we always start by seeding the empty e-graph with the pairwise disequality of all the numerals.

Experiment 1. As we are interested in finding whether a set of (dis)equalities is consistent, we benchmark the time it takes to check the consistency of the e-graph while varying the ratio of disequalities. More precisely, for the Equality Embedding method, we benchmark the execution time of Line 29 of Listing 1, and for the Disequality Edges method, we benchmark the execution time of the `is_consistent()` function of Listing 7. The results are in Figure 2.

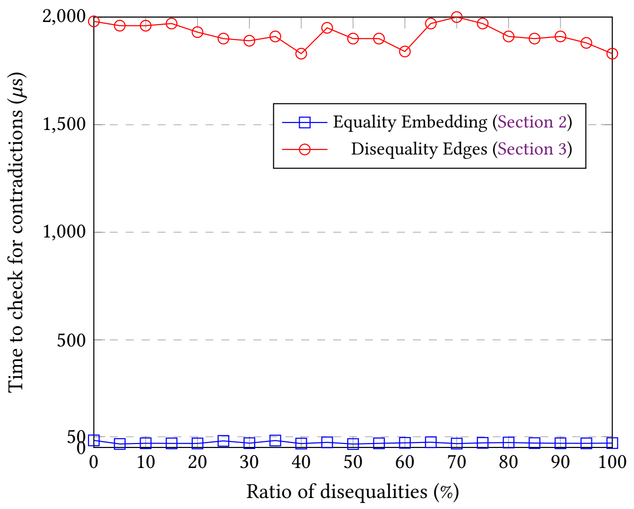


Figure 2. Time to deduce the consistency of an e-graph seeded with 30 K pairs of expressions with a given disequalities/equalities ratio.

The execution time of both methods is generally constant where Equality Embedding hovers around 20 μs while Disequality Edges hovers around 1.9 ms. Thus, Equality Embedding is two orders of magnitude faster than Disequality Edges. We attribute this difference to the fact that Disequality Edges' consistency check must traverse the forbid lists of every class and look up for each identifier the root one in the union-find data structure, while Equality Embedding only checks the root identifier of two e-classes.

Experiment 2. One difference between Equality Embedding (Section 2) and Disequality Edges (Section 3) is that the former requires performing e-matching [13] and saturating the e-graph – both expensive operations – while the former does not. Therefore, starting with an e-graph containing only the disequalities between the numerals, we benchmark, for both methods, the time for building the whole e-graph and checking the consistency of the resulting graph. We plot the results in Figure 3.

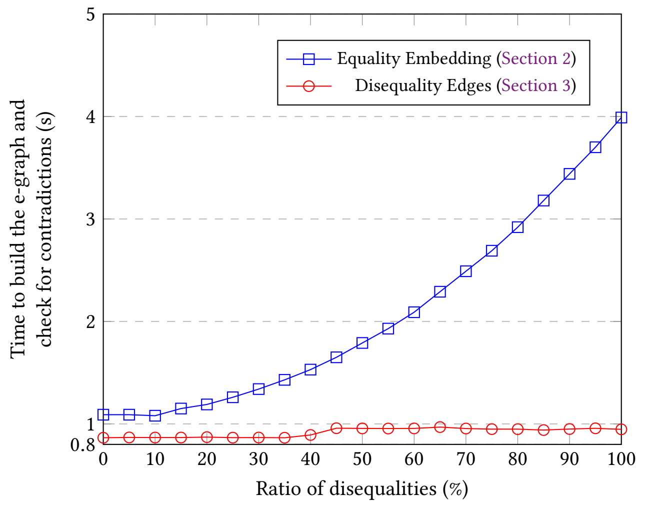


Figure 3. The time to build and deduce the consistency of an e-graph seeded with 30 K pairs of expressions a given disequalities/equalities ratio.

The time for the Equality Embedding method increases from 1 s to 4 s, and from 0.86 s to 0.96 s for the Disequality Edges method. As expected, the cost of performing e-matching and saturating the e-graph becomes apparent. While the Equality Embedding method seems to execute in constant time, in reality it is not: we observe a subtle increase in run time which we attribute to the increasing length of each node's forbid list. As the number of disequalities increases, more classes are added to forbid lists, and the method copies forbid lists from one e-class into the other at every union.

Experiment 3. Saturation in Equality Embedding introduces additional nodes to the e-graph which in turn introduces additional e-classes. For example, every equality term is duplicated with the arguments swapped as a consequence of the symmetry of equality. On the other hand, Disequality Edges introduces no new nodes and no new e-classes. To this end, we plot the number of e-classes and nodes in the e-graph of all trials in Figure 4.

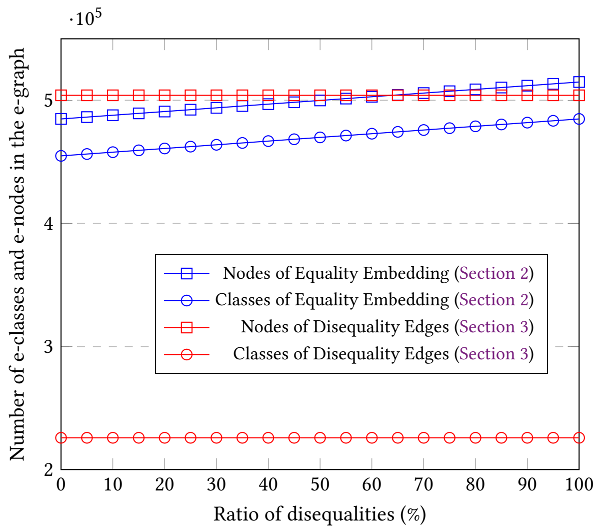


Figure 4. The number of e-classes (circles) and e-nodes (circle) of the e-graph seeded with 30 K pairs of expressions a given disequalities/equalities ratio.

As expected, the numbers of e-classes and e-nodes in the Equality Embedding method increase while those of Disequality Edges are constant. Also, we observe that the number of e-classes in the Equality Embedding is much greater than the one in Disequality Edges and is only slightly smaller than the number e-nodes. One consequence is that e-classes in the Equality Embedding method are small: they do not contain many nodes and thus the power of the resulting e-graph becomes questionable. We attribute this difference to the fact that the Equality Embedding method synthesizes many terms, each necessitating their own e-classes.

4.1 Results

Our experiments show that checking the consistency of an e-graph using the Equality Embedding method is two-orders of magnitude faster than the Disequality Edges method. Yet, this speed-up is shadowed by the expensive process of building the e-graph, whether through fixing to maintain its invariant or through saturation. Moreover, e-graphs are incremental data structures; rebuilding is a common and frequent operation. Thus, this operation dominates the overall run time, rendering Equality Embedding less efficient then Disequality Edges.

We believe that we can further improve our implementation of `is_consistent()` from Listing 7: While unioning two e-classes, the check for consistency can be performed and cached in a field in the e-graph which `is_consistent()` can simply return.

5 Conclusion and Future Work

Our work on Propel [11, 12] suggests that disequalities can be beneficial in theorem provers. Also, the original work on e-graphs accounted for disequalities [7]. In this work, we present preliminary results on integrating e-graphs into the egg high-performance e-graph library. Our initial results indicate that Disequality Edges (Method 2, Section 3) may be more efficient for our use case. We plan to continue our evaluation and our work to efficiently support disequalities in e-graphs and integrate such data structure into the Propel theorem prover. We believe that our extensions to egg could already be valuable to other researchers and we plan to release them to the community.

References

[1] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–153. https://doi.org/10.1007/978-3-642-12002-2_12

[2] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[3] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

[4] egg Documentation. 2019. *Pattern in egg*. https://docs.rs/egg/latest/egg/struct.Pattern.html Accessed: 2024-04-12.

[5] egg Documentation. 2019. *SymbolLang in egg*. https://docs.rs/egg/latest/egg/struct.SymbolLang.html Accessed: 2024-04-12.

[6] Bernard A. Galler and Michael J. Fisher. 1964. An Improved Equivalence Algorithm. *Commun. ACM* 7, 5 (May 1964), 301–303. https://doi.org/10.1145/364099.364331

[7] Charles Gregory Nelson. 1980. *Techniques for Program Verification*. Ph. D. Dissertation. Stanford, CA, USA.

[8] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Term Rewriting and Applications*, Jürgen Giesl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 453–468.

[9] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

[10] 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, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

[11] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 162 (June 2023), 24 pages. https://doi.org/10.1145/3591276

[12] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 178 (June 2024), 24 pages. https://doi.org/10.1145/3656408

[13] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 35 (Jan. 2022), 22 pages. https://doi.org/10.1145/3498696