Fast Approximations of Quantifier Elimination

Isabel Garcia-Contreras[1][0000-0001-6098-3895], Hari Govind V K[1][0000-0002-2789-5997], Sharon Shoham[2][0000-0002-7226-3526], and Arie Gurfinkel[1][0000-0002-5964-6792]

[1] University of Waterloo, Waterloo, Canada
{igarciac,hgvedira,agurfink}@uwaterloo.ca
[2] Tel-Aviv University, Tel Aviv, Israel
sharon.shoham@cs.tau.ac.il

Abstract. Quantifier elimination (qelim) is used in many automated reasoning tasks including program synthesis, exist-forall solving, quantified SMT, Model Checking, and solving Constrained Horn Clauses (CHCs). Exact qelim is computationally expensive. Hence, it is often approximated. For example, Z3 uses "light" pre-processing to reduce the number of quantified variables. CHC-solver Spacer uses model-based projection (MBP) to under-approximate qelim relative to a given model, and over-approximations of qelim can be used as abstractions.
In this paper, we present the QEL framework for fast approximations of qelim. QEL provides a uniform interface for both quantifier reduction and model-based projection. QEL builds on the egraph data structure – the core of the EUF decision procedure in SMT – by casting quantifier reduction as a problem of choosing *ground* (i.e., variable-free) representatives for equivalence classes. We have used QEL to implement MBP for the theories of Arrays and Algebraic Data Types (ADTs). We integrated QEL and our new MBP in Z3 and evaluated it within several tasks that rely on quantifier approximations, outperforming state-of-the-art.

1 Introduction

Quantifier Elimination (qelim) is used in many automated reasoning tasks including program synthesis [18], exist-forall solving [8,9], quantified SMT [5], and Model Checking [17]. Complete qelim, even when possible, is computationally expensive, and solvers often approximate it. We call these approximations *quantifier reductions*, to separate them from qelim. The difference is that quantifier reduction might leave some free variables in the formula.

For example, Z3 [19] performs quantifier reduction, called QeLite, by greedily substituting variables by definitions syntactically appearing in the formulas. While it is very useful, it is necessarily sensitive to the order in which variables are substituted and depends on definitions appearing explicitly in the formula. Even though it may seem that these shortcomings need to be tolerated to keep QeLite fast, in this paper we show that it is not actually the case; we propose an egraph-based algorithm, QEL, to perform fast quantifier reduction that is complete relative to some semantic properties of the formula.

Egraph [20] is a data structure that compactly represents infinitely many terms and their equivalence classes. It was initially proposed as a decision procedure for EUF [20] and used for theorem proving (e.g., SIMPLIFY [7]). Since then, the applications of egraphs have grown. Egraphs are now used as term rewrite systems in equality saturation [15,23], for theory combination in SMT solvers [21,7], and for term abstract domains in Abstract Interpretation [12,6,10].

Using egraphs for rewriting or other formula manipulations (like qelim) requires a special operation, called *extract*, that converts nodes in the egraph back into terms. Term extraction was not considered when egraphs were first designed [20]. As far as we know, extraction was first studied in the application of egraphs for compiler optimization. Specifically, equality saturation [15,22] is an optimization technique over egraphs that consists in populating an egraph with many equivalent terms inferred by applying rules. When the egraph is saturated, i.e., applying the rules has no effect, the equivalent term that is most desired, e.g., smallest in size, is *extracted*. This is a recursive process that extracts each sub-term by choosing one representative among its equivalents.

Application of egraphs to rewriting have recently resurged driven by the `egg` library [24] and the associated workshop[3]. In [24], the authors show, once again, the power and versatility of this data structure. Motivated by applications of equality saturation, they provide a generic and efficient framework equipped with term extraction, based on an extensible class analysis.

Egraphs seem to be the perfect data-structure to address the challenges of quantifier reduction: they allow reasoning about infinitely many equivalent terms and consider all available variable definitions and orderings at once. However, things are not always what they appear. The key to quantifier reduction is finding ground (i.e., variable-free) representatives for equivalence classes with free variables. This goes against existing techniques for term extraction since it requires selecting larger, rather than smaller, terms to be representatives. Selecting representatives carelessly makes term extraction diverge. To our surprise, this problem has not been studied so far. In fact, `egg` [24] incorrectly claims that any representative function can be used with its term extraction, while the implementation diverges. In this paper, we bridge this gap by providing necessary and sufficient conditions for a representative function to be admissible for term extraction as defined in [15,24]. Furthermore, we extend extraction from terms to formulas to enable extracting a formula of the egraph.

Our main contribution is a new quantifier reduction algorithm, called QEL. Building on the term extraction described above, it is formulated as finding a representative function that maximizes the number of ground terms as representatives. Furthermore, it greedily attempts to represent variables without ground representatives in terms of other variables, thus further reducing the number of variables in the output. We show that QEL is complete relative to ground definitions entailed by the formula. Specifically, QEL guarantees to eliminate a variable if it is equivalent to a ground term.

---
[3] https://pldi22.sigplan.org/series/egraphs.

Whenever an application requires eliminating all free variables, incomplete techniques such as QeLite or QEL are insufficient. In this case, qelim is under-approximated using a Model-based Projection (MBP) that uses a model M of a formula to guide under-approximation using equalities and variable definitions that are consistent with M. In this paper, we show that MBP can be implemented using our new techniques for QEL together with the machinery from equality saturation. Just like SMT solvers use egraphs as glue to combine different theory solvers, we use egraphs as glue to combine projection for different theories. In particular, we give an algorithm for MBP in the combined theory of Arrays and Algebraic DataTypes (ADTs). The algorithm uses insights from QEL to produce less under-approximate MBPs.

We implemented QEL and the new MBP using egraphs inside the state-of-art SMT solver Z3 [19]. Our implementation (referred to as Z3eg) replaces the existing QeLite and MBP. We evaluate our algorithms in two contexts. First, inside the QSAT [5] algorithm for quantified satisfiability. The performance of QSAT in Z3eg is improved, compared to QSAT in Z3, when ADTs are involved. Second, we evaluate our algorithms inside the Constrained Horn Clause (CHC) solver Spacer [17]. Our experiments show that Spacer in Z3eg solves many more benchmarks containing nested Arrays and ADTs.

Related Work. Quantifier reduction by variable substitution is widely used in quantified SMT [11,5]. To our knowledge, we are the first to look at this problem semantically and provide an algorithm that guarantees that the variable is eliminated if the formula entails that it has a ground definition.

Term extraction for egraphs comes from equality saturation [15,22]. The `egg` Rust library [24] is a recent implementation of equality saturation that supports rewriting and term extraction. However, we did not use `egg` because we integrated QEL within Z3 and built it using Z3 data structures instead.

Model-based projection was first introduced for the Spacer CHC solver for LIA and LRA [17] and extended to the theory of Arrays [16] and ADTs [5]. Until now, it was implemented by syntactic rewriting. Our egraph-based MBP implementation is less sensitive to syntax and, more importantly, allows for combining MBPs of multiple theories for MBP of the combination. As a result, our MBP is more general and less model dependent. Specifically, it requires fewer model equalities and produces more general under-approximations than [16,5].

Outline. The rest of the paper is organized as follows. Sec. 2 provides background. Sec. 3 introduces term extraction, extends it to formulas, and characterizes representative-based term extraction for egraphs. Sec. 4 presents QEL, our algorithm for fast quantifier reduction that is relatively complete. Sec. 5 shows how to compute MBP combining equality saturation and the ideas from Sec. 4 for the theories of ADTs and Arrays. All algorithms have been implemented in Z3 and evaluated in Sec. 6.

2 Background

We assume the reader is familiar with multi-sorted first-order logic (FOL) with equality and the theory of equality with uninterpreted functions (EUF) (for an introduction see, e.g. [4]). We use $\approx$ to denote the designated logical equality symbol. For simplicity of presentation, we assume that the FOL signature Σ contains only functions (i.e., no predicates) and constants (i.e., 0-ary functions). To represent predicates, we assume the FOL signature has a designated sort Bool, and two Bool constants $\top$ and $\bot$, representing true, and false respectively. We then use Bool-valued functions to represent predicates, using $P(a) \approx \top$ and $P(a) \approx \bot$ to mean that $P(a)$ is true or false, respectively. Informally, we continue to write $P(a)$ and $\neg P(a)$ as a syntactic sugar for $P(a) \approx \top$ and $P(a) \approx \bot$, respectively. We use lowercase letters like a, b for constants, and f, g for functions, and uppercase letters like P, Q for Bool functions that represent predicates. We denote by $\psi^{\exists}$ the existential closure of ψ.

Quantifier Elimination (qelim). Given a quantifier-free (QF) formula φ with free variables $\boldsymbol{v}$, *quantifier elimination* of $\varphi^{\exists}$ is the problem of finding a QF formula ψ with no free variables such that $\psi \equiv \varphi^{\exists}$. For example, a qelim of $\exists a \cdot (a \approx x \land f(a) > 3)$ is $f(x) > 3$; and, there is no qelim of $\exists x \cdot (f(x) > 3)$, because it is impossible to restrict f to have "at least one value in its range that is greater than 3" without a quantifier.

Model Based Projection (MBP). Let φ be a formula with free variables $\boldsymbol{v}$, and M a model of φ. A *model-based projection* of φ relative to M is a QF formula ψ such that $\psi \Rightarrow \varphi^{\exists}$ and $M \models \psi$. That is, ψ has no free variables, is an under-approximation of φ, and satisfies the designated model M, just like φ. MBP is used by many algorithms to under-approximate qelim, when the computation of qelim is too expensive or, for some reason, undesirable.

Egraphs. An egraph is a well-known data structure to compactly represent a set of terms and an equivalence relation on those terms [20]. Throughout the paper, we assume that graphs have an ordered successor relation and use $n[i]$ to denote the ith successor (child) of a node n. An out-degree of a node n, $\mathtt{deg}(n)$, is the number of edges leaving n. Given a node n, $\mathtt{parents}(n)$ denotes the set of nodes with an outgoing edge to n and $\mathtt{children}(n)$ denotes the set of nodes with an incoming edge from n.

Definition 1. *Let Σ be a first-order logic signature. An* egraph *is a tuple $G = \langle N, E, L, \mathtt{root} \rangle$, where*

(a) $\langle N, E \rangle$ is a directed acyclic graph,
(b) L maps nodes to function symbols in Σ or logical variables, and
(c) $\mathtt{root} : N \mapsto N$ maps a node to its root *such that the relation $\rho_{\mathtt{root}} \triangleq \{(n, n') \mid \mathtt{root}(n) = \mathtt{root}(n')\}$ is an equivalence relation on N that is* closed under congruence*: $(n, n') \in \rho_{\mathtt{root}}$ whenever n and n' are* congruent *under $\mathtt{root}$, i.e., whenever $L(n) = L(n')$, $\mathtt{deg}(n) = \mathtt{deg}(n') > 0$, and, $\forall 1 \leq i \leq \mathtt{deg}(n) \cdot (n[i], n'[i]) \in \rho_{\mathtt{root}}$.*

$$\varphi_1(x, y, z) \triangleq z \approx read(a, x) \wedge k + 1 \approx read(a, y) \wedge x \approx y \wedge 3 > z$$

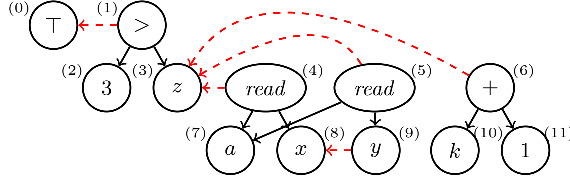


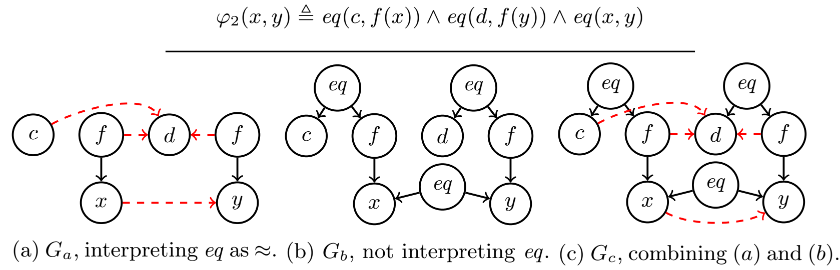
Fig. 1: Example egraph of φ_1.

Given an egraph G, the *class* of a node $n \in G$, $class(n) \triangleq \rho_{\texttt{root}}(n)$, is the set of all nodes that are equivalent to n. The term of n, $term(n)$, with $L(n) = f$ is f if $\texttt{deg}(n) = 0$ and $f(term(n[1]), \ldots, term(n[\texttt{deg}(n)]))$, otherwise. We assume that the terms of different nodes are different, and refer to a node n by its term.

An example of an egraph $G = \langle N, E, L, \texttt{root}\rangle$ is shown in Fig. 1. A symbol f inside a circle depicts a node n with label $L(n) = f$, solid black and dashed red arrows depict E and `root`, respectively. The order of the black arrows from left to right defines the order of the children. In our examples, we refer to a specific node i by its number using $\mathsf{N}(i)$ or its term, e.g., $\mathsf{N}(k+1)$. A node n without an outgoing red arrow is its own root. A set of nodes connected to the same node with red edges forms an equivalence class. In this example, `root` defines the equivalence classes $\{\mathsf{N}(3), \mathsf{N}(4), \mathsf{N}(5), \mathsf{N}(6)\}$, $\{\mathsf{N}(8), \mathsf{N}(9)\}$, and a class for each of the remaining nodes. Examples of some terms in G are $term(\mathsf{N}(9)) = y$ and $term(\mathsf{N}(5)) = read(a, y)$.

An Egraph of a Formula. We consider formulas that are conjunctions of equality literals (recall that we represent predicate applications by equality literals). Given a formula $\varphi \triangleq (t_1 \approx u_1 \wedge \cdots \wedge t_k \approx u_k)$, an egraph from φ is built (following the standard procedure [20]) by creating nodes for each t_i and u_i, recursively creating nodes for their subexpressions, and merging the classes of each pair t_i and u_i, computing the congruence closure for `root`. We write $egraph(\varphi)$ for an egraph of φ constructed via some deterministic procedure based on the recipe above. Fig. 1 shows an $egraph(\varphi_1)$ of φ_1. The equality $z \approx read(a, x)$ is captured by $\mathsf{N}(3)$ and $\mathsf{N}(4)$ belonging to the same class (i.e., red arrow from $\mathsf{N}(4)$ to $\mathsf{N}(3)$). Similarly, the equality $x \approx y$ is captured by a red arrow from $\mathsf{N}(9)$ to $\mathsf{N}(8)$. Note that by congruence, φ_1 implies $read(a, x) \approx read(a, y)$, which, by transitivity, implies that $k + 1 \approx read(a, x)$. In Fig. 1, this corresponds to red arrows from $\mathsf{N}(5)$ and $\mathsf{N}(6)$ to $\mathsf{N}(3)$. The predicate application $3 > z$ is captured by the red arrow from $\mathsf{N}(1)$ to $\mathsf{N}(0)$. From now on, we omit $\top$ and $\bot$ and the corresponding edges from figures to avoid clutter.

Explicit and Implicit Equality. Note that egraphs represent equality implicitly by placing nodes with equal terms in the same equivalence class. Sometimes, it is necessary to represent equality explicitly, for example, when using egraphs for equality-aware rewriting (e.g., in `egg` [24]). To represent equality explicitly, we



(a) G_a, interpreting eq as $\approx$. (b) G_b, not interpreting eq. (c) G_c, combining (a) and (b).

Fig. 2: Different egraph interpretations for φ_2.

introduce a binary Bool function eq and write $eq(a, b)$ for an equality that has to be represented explicitly. We change the *egraph* algorithm to treat $eq(a, b)$ as both a function application, and as a logical equality $a \approx b$: when processing term $eq(a, b)$, the algorithm both adds $eq(a, b)$ to the egraph, and merges the nodes for a and b into one class. For example, Fig. 2 shows three different interpretations of a formula φ_2 with equality interpreted: implicitly (as in [20]), explicitly (as in [24]), and both implicitly and explicitly (as in this paper).

3 Extracting Formulas from Egraphs

Egraphs were proposed as a decision procedure for EUF [20] – a setting in which converting an egraph back to a formula, or *extracting*, is irrelevant. Term extraction has been studied in the context of equality saturation and term rewriting [15,24]. However, existing literature presents extraction as a heuristic, and, to the best of our knowledge, has not been exhaustively explored. In this section, we fill these gaps in the literature and extend extraction from terms to formulas.

Term Extraction. We begin by recalling how to extract the term of a node. The function `ntt` (node-to-term) in Fig. 3 does an extraction parametrized by a representative function $\texttt{repr} : N \mapsto N$ (same as in [24]). A function `repr` assigns each class a unique representative node (i.e., nodes in the same class are mapped to the same representative) so that $\rho_{\texttt{root}} = \rho_{\texttt{repr}}$. The function `ntt` extracts a term of a node recursively, similarly to *term*, except that the representatives of the children of a node are used instead of the actual children. We refer to terms built in this way by $\texttt{ntt}(n, \texttt{repr})$ and omit `repr` when it is clear from the context.

As an example, consider $\texttt{repr}_1 \triangleq \{\mathsf{N}(3), \mathsf{N}(8))\}$ for Fig. 1. For readability, we denote representative functions by sets of nodes that are the class representatives, omitting $\mathsf{N}(\top)$ that always represents its class, and omitting all singleton classes. Thus, $\texttt{repr}_1$ maps all nodes in $class(\mathsf{N}(3))$ to $\mathsf{N}(3)$, nodes in $class(\mathsf{N}(8))$ to $\mathsf{N}(8)$, nodes in $class(\mathsf{N}(\top))$ to $\mathsf{N}(\top)$, and all singleton classes to themselves. For example, $\texttt{ntt}(\mathsf{N}(5))$ extracts $read(a, x)$, since $\mathsf{N}(9)$ has as representative $\mathsf{N}(8)$.

Formula Extraction. Let $G = egraph(\varphi)$ be an egraph of some formula φ. A formula ψ is a *formula of* G, written $isFormula(G, \psi)$, if $\psi^\exists \equiv \varphi^\exists$.

```
egraph :: to_formula(repr, S)
 1: Lits := ∅
 2: for r = repr(r) ∈ N do
 3:   t := ntt(r, repr)
 4:   for n ∈ (class(r) \ r) do
 5:     if n ∉ S then
 6:       Lits := Lits ∪ {t ≈ ntt(n, repr)}
 7: ret ⋀ Lits

egraph :: ntt(n, repr)
 8: f := L[n]
 9: if deg(n) = 0 then
10:   ret f
11: else
12:   for i ∈ [1, deg(n)] do
13:     Args[i] := ntt(repr(n[i]), repr)
14:   ret f(Args)
```

Fig. 3: Producing formulas from an egraph.

Fig. 3 shows an algorithm $\texttt{to_formula}(\texttt{repr}, S)$ to compute a formula ψ that satisfies $\mathit{isFormula}(G, \psi)$ for a given egraph G. In addition to `repr`, `to_formula` is parameterized by a set of nodes $S \subseteq N$ to exclude[4]. To produce the equalities corresponding to the classes, for each representative r, for each $n \in (\mathit{class}(r) \setminus \{r\})$ the output formula has a literal $\texttt{ntt}(r) \approx \texttt{ntt}(n)$. For example, using $\texttt{repr}_1$ for the egraph in Fig. 1, we obtain for $\mathit{class}(\mathsf{N}(8))$, $(x \approx y)$; for $\mathit{class}(\mathsf{N}(3))$, $(z \approx \mathit{read}(a, x) \land z \approx \mathit{read}(a, x) \land z \approx k+1)$; and for $\mathit{class}(\mathsf{N}(0))$, $(\top \approx 3 > z)$. The final result (slightly simplified) is: $x \approx y \land z \approx \mathit{read}(a, x) \land z \approx k + 1 \land 3 > z$.

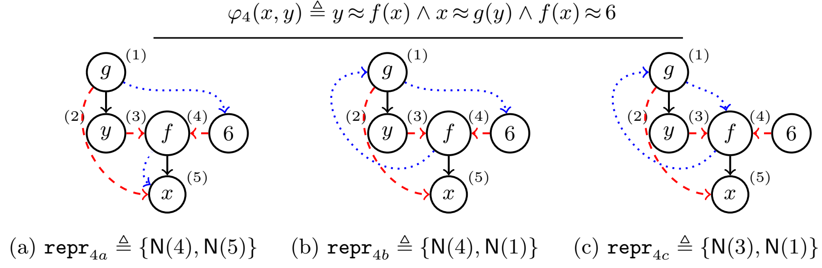
Let $G = \mathit{egraph}(\varphi)$ for some formula φ. Note that, ψ computed by `to_formula` is not syntactically the same as φ. That is, `to_formula` is not an inverse of *egraph*. Furthermore, since `to_formula` commits to one representative per class, it is limited in what formulas it can generate. For example, since $x \approx y$ is in φ_1, for any `repr`, φ_1 cannot be the result of `to_formula`, because the output can contain only one of $\mathit{read}(a, x)$ or $\mathit{read}(a, y)$.

Representative Functions. The representative function is instrumental for determining the terms that appear in the extracted formula. To illustrate the importance of representative choice, consider the formula φ_4 of Fig. 4 and its egraph $G_4 = \mathit{egraph}(\varphi_4)$. For now, ignore the blue dotted lines. For $\texttt{repr}_{4a}$, `to_formula` obtains $\psi_a \triangleq (x \approx g(6) \land f(x) \approx 6 \land y \approx 6)$. For $\texttt{repr}_{4b}$, `to_formula` produces $\psi_b \triangleq (g(6) \approx x \land f(g(6)) \approx 6 \land y \approx 6)$. In some applications (like qelim considered in this paper) ψ_b is preferred to ψ_a: simply removing the literals $g(6) \approx x$ and $y \approx 6$ from ψ_b results in a formula equivalent to $\exists x, y \cdot \varphi_4$ that does not contain variables. Consider a third representative choice $\texttt{repr}_{4c}$, for node $\mathsf{N}(1)$, `ntt` does not terminate: to produce a term for $\mathsf{N}(1)$, a term for $\mathsf{N}(3)$, the representative of its child, $\mathsf{N}(2)$, is required. Similarly to produce a term for $\mathsf{N}(3)$, a term for the representative of its child node $\mathsf{N}(5)$, $\mathsf{N}(1)$, is necessary. Thus, none of the terms can be extracted with $\texttt{repr}_{4c}$.

For extraction, representative functions `repr` are either provided explicitly or implicitly (as in [24]), the latter by associating a cost to nodes and/or terms and letting the representative be a node with minimal cost. However, observe that not all costs guarantee that the chosen `repr` can be used (the computation does not terminate). For example, the ill-defined $\texttt{repr}_{4c}$ from above is a representative function that satisfies the cost function that assigns function applications cost 0

[4] The set S affects the result, but for this section, we restrict to the case of $S \triangleq \emptyset$.

$\varphi_4(x, y) \triangleq y \approx f(x) \wedge x \approx g(y) \wedge f(x) \approx 6$



(a) $\texttt{repr}_{4a} \triangleq \{\mathsf{N}(4), \mathsf{N}(5)\}$ (b) $\texttt{repr}_{4b} \triangleq \{\mathsf{N}(4), \mathsf{N}(1)\}$ (c) $\texttt{repr}_{4c} \triangleq \{\mathsf{N}(3), \mathsf{N}(1)\}$

Fig. 4: Egraphs of φ_4 with $G_{\texttt{repr}}$.

and variables and constants cost 1. A commonly used cost function is term AST size, which is sufficient to ensure termination of $\texttt{ntt}(n, \texttt{repr})$.

We are thus interested in characterizing representative functions motivated by two observations: not every cost function guarantees that $\texttt{ntt}(n)$ terminates; and the kind of representative choices that are most suitable for qelim ($\texttt{repr}_{4b}$) cannot be expressed over term AST size.

Definition 2. *Given an egraph* $G = \langle N, E, L, \texttt{root}\rangle$, *a representative function* $\texttt{repr} : N \to N$ *is* admissible for G *if*

(a) $\texttt{repr}$ *assigns a unique representative per class,*
(b) $\rho_{\texttt{root}} = \rho_{\texttt{repr}}$*, and*
(c) the graph $G_{\texttt{repr}}$ *is acyclic, where* $G_{\texttt{repr}} = \langle N, E_{\texttt{repr}}\rangle$ *and* $E_{\texttt{repr}} \triangleq \{(n, \texttt{repr}(c)) \mid c \in \textit{children}(n), n \in N\}$.

Dotted blue edges in the graphs of Fig. 4 show the corresponding $G_{\texttt{repr}}$. Intuitively, for each node n, all reachable nodes in $G_{\texttt{repr}}$ are the nodes whose $\texttt{ntt}$ term is necessary to produce the $\texttt{ntt}(n)$. Observe that $G_{\texttt{repr}_{4c}}$ has a cycle, thus, $\texttt{repr}_{4c}$ is not admissible.

Theorem 1. *Given an egraph* G *and a representative function* $\texttt{repr}$*, the function* $G.\texttt{to_formula}(\texttt{repr}, \emptyset)$ *terminates with result* ψ *such that* $\textit{isFormula}(G, \psi)$ *iff* $\texttt{repr}$ *is admissible for* G*.*

To the best of our knowledge, Theorem 1 is the first complete characterization of all terms of a node that can be obtained by extraction based on class representatives (via describing all admissible $\texttt{repr}$, note that the number is finite). This result contradicts [24], where it is claimed to be possible to extract a term of a node for any cost function. The counterexample is $\texttt{repr}_{4c}$. Importantly, this characterization allows us to explore representative functions outside those in the existing literature, which, as we show in the next section, is key for qelim.

4 Quantifier Reduction

Quantifier reduction is a relaxation of quantifier elimination: given two formulas φ and ψ with free variables $\boldsymbol{v}$ and $\boldsymbol{u}$, respectively, ψ is a *quantifier reduction* of

Input: A formula φ with free variables $\boldsymbol{v}$.
Output: A quantifier reduction of φ.

```
QEL(φ, v)
1: G := egraph(φ)
2: repr := G.find_defs(v)
3: repr := G.refine_defs(repr, v)
4: core := G.find_core(repr)
5: ret G.to_formula(repr, G.Nodes() \ core)
```

Algorithm 1: QEL – Quantifier reduction using egraphs.

φ if $\boldsymbol{u} \subseteq \boldsymbol{v}$ and $\varphi^{\exists} \equiv \psi^{\exists}$. If $\boldsymbol{u}$ is empty, then ψ is a quantifier elimination of $\varphi^{\exists}$. Note that quantifier reduction is possible even when quantifier elimination is not (e.g., for EUF). We are interested in an efficient quantifier reduction algorithm (that can be used as pre-processing for qelim), even if a complete qelim is possible (e.g., for LIA). In this section, we present such an algorithm called QEL.

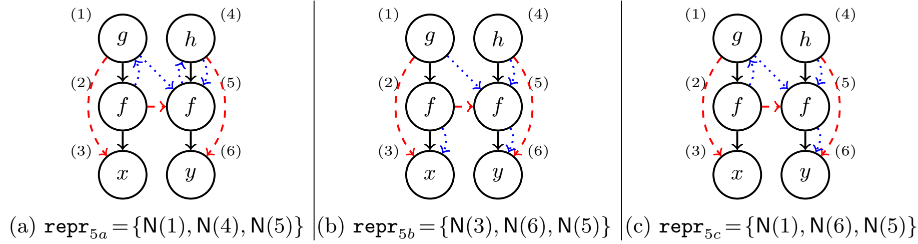
Intuitively, QEL is based on the well-known substitution rule: $(\exists x \cdot x \approx t \wedge \varphi) \equiv \varphi[x \mapsto t]$. A naive implementation of this rule, called QeLite in Z3, looks for syntactic definitions of the form $x \approx t$ for a variable x and an x-free term t and substitutes x with t. While efficient, QeLite is limited because of: (a) dependence on syntactic equality in the formula (specifically, it misses implicit equalities due to transitivity and congruence); (b) sensitivity to the order in which variables are eliminated (eliminating one variable may affect available syntactic equalities for another); and (c) difficulty in dealing with circular equalities such as $x \approx f(x)$.

For example, consider the formula $\varphi_4(x, y)$ in Fig. 4. Assume that y is eliminated first using $y \approx f(x)$, resulting in $x \approx g(f(x)) \wedge f(x) \approx 6$. Now, x cannot be eliminated since the only equality for x is circular. Alternatively, assume that QeLite somehow noticed that by transitivity, φ_4 implies $y \approx 6$, and obtains $(\exists y \cdot \varphi_4) \triangleq x \approx g(6) \wedge f(x) \approx 6$. This time, $x \approx g(6)$ can be used to obtain $f(g(6)) \approx 6$ that is a qelim of $\varphi_4^{\exists}$. Thus, both the elimination order and implicit equalities are crucial.

In QEL, we address the above issues by using an egraph data structure to concisely capture all implicit equalities and terms. Furthermore, egraphs allow eliminating multiple variables together, ensuring that a variable is eliminated if it is equivalent (explicitly or implicitly) to a ground term in the egraph.

Pseudocode for QEL is shown in Alg. 1. Given an input formula φ, QEL first builds its egraph G (line 1). Then, it finds a representative function `repr` that maps variables to equivalent ground terms, as much as possible (line 2). Next, it further reduces the remaining free variables by refining `repr` to map each variable x to an equivalent x-free (but not variable-free) term (line 3). At this point, QEL is committed to the variables to eliminate. To produce the output, `find_core` identifies the subset of the nodes of G, which we call *core*, that must be considered in the output (line 4). Finally, `to_formula` converts the core of G to the resulting formula (line 5). We show that the combination of these steps is even stronger than variable substitution.

$$\varphi_5(x,y) \triangleq x \approx g(f(x)) \land y \approx h(f(y)) \land f(x) \approx f(y)$$



(a) $\texttt{repr}_{5a} = \{\mathsf{N}(1), \mathsf{N}(4), \mathsf{N}(5)\}$ (b) $\texttt{repr}_{5b} = \{\mathsf{N}(3), \mathsf{N}(6), \mathsf{N}(5)\}$ (c) $\texttt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$

Fig. 5: Egraphs including $G_{\texttt{repr}}$ of φ_5.

To illustrate QEL, we apply it on φ_1 and its egraph G from Fig. 1. The function `find_defs` returns $\texttt{repr} = \{\mathsf{N}(6), \mathsf{N}(8)\}$[5]. Node $\mathsf{N}(6)$ is the only node with a ground term in the equivalence class $\mathit{class}(\mathsf{N}(3))$. This corresponds to the definition $z \approx k+1$. Node $\mathsf{N}(8)$ is chosen arbitrarily since $\mathit{class}(\mathsf{N}(8))$ has no ground terms. There is no refinement possible, so `refine_defs` returns `repr`. The core is $N \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. Nodes $\mathsf{N}(3)$ and $\mathsf{N}(9)$ are omitted because they correspond to variables with definitions (under `repr`), and $\mathsf{N}(5)$ is omitted because it is congruent to $\mathsf{N}(4)$ so only one of them is needed. Finally, `to_formula` produces $k+1 \approx \mathit{read}(a,x) \land 3 > k+1$. Variables z and y are eliminated.

In the rest of this section we present QEL in detail and QEL's key properties.

Finding Ground Definitions. Ground variable definitions are found by selecting a representative function `repr` that ensures that the maximum number of terms in the formula are rewritten into ground equivalent ones, which, in turn, means finding a ground definition for all variables that have one.

Computing a representative function `repr` that is admissible and ensures finding ground definitions when they exist is not trivial. Naive approaches for identifying ground terms, such as iterating arbitrarily over the classes and selecting a representative based on $\mathit{term}(n)$ are not enough – $\mathit{term}(n)$ may not be in the output formula. It is also not possible to make a choice based on $\texttt{ntt}(n)$, since, in general, it cannot be yet computed (`repr` is not known yet).

Admissibility raises an additional challenge since choosing a node that appears to be a definition (e.g., not a leaf) may cause cycles in $G_{\texttt{repr}}$. For example, consider φ_5 of Fig. 5. Assume that $\mathsf{N}(1)$ and $\mathsf{N}(4)$ are chosen as representatives of their equivalence classes. At this point, $G_{\texttt{repr}}$ has two edges: $\langle \mathsf{N}(5), \mathsf{N}(4)\rangle$ and $\langle \mathsf{N}(2), \mathsf{N}(1)\rangle$, shown by blue dotted lines in Fig. 5a. Next, if either $\mathsf{N}(2)$ or $\mathsf{N}(5)$ are chosen as representatives (the only choices in their class), then $G_{\texttt{repr}}$ becomes cyclic (shown in blue in Fig. 5a). Furthermore, backtracking on representative choices needs to be avoided if we are to find a representative function efficiently.

Alg. 2 finds a representative function `repr` while overcoming these challenges. To ensure that the computed representative function is admissible (without back-

[5] Recall that we only show representatives of non-singleton classes.

```
egraph :: find_defs(v)
1: for n ∈ N do repr(n) := ★
2: todo := {leaf(n) | n ∈ N ∧ ground(n)}
3: repr := process(repr, todo)
4: todo := {leaf(n) | n ∈ N}
5: repr := process(repr, todo)
6: ret repr
```

```
egraph :: process(repr, todo)
7: while todo ≠ ∅ do
8:   n := todo.pop()
9:   if repr(n) ≠ ★ then continue
10:  for n′ ∈ class(n) do repr(n′) := n
11:  for n′ ∈ class(n) do
12:    for p ∈ parents(n′) do
13:      if ∀c ∈ children(p) · repr(c) ≠ ★ then
14:        todo.push(p)
15: ret repr
```

Algorithm 2: Find definitions maximizing groundness.

tracking), Alg. 2 selects representatives for each class using a “bottom up” approach. Namely, leaves cannot be part of cycles in $G_{\texttt{repr}}$ because they have no outgoing edges. Thus, they can always be safely chosen as representatives. Similarly, a node whose children have already been assigned representatives in this way (leaves initially), will also never be part of a cycle in $G_{\texttt{repr}}$. Therefore, these nodes are also safe to be chosen as representatives.

This intuition is implemented in `find_defs` by initializing `repr` to be undefined (★) for all nodes, and maintaining a workset, *todo*, containing nodes that, if chosen for the remaining classes (under the current selection), maintain acyclicity of $G_{\texttt{repr}}$. The initialization of *todo* includes leaves only. The specific choice of leaves ensures that ground definitions are preferred, and we return to it later. After initialization, the function `process` extracts an element from *todo* and sets it as the representative of its class if the class has not been assigned yet (lines 9 and 10). Once a class representative has been chosen, on lines 11 to 14, the parents of all the nodes in the class such that all the children have been chosen (the condition on line 13) are added to *todo*.

So far, we discussed how admissibility of `repr` is guaranteed. To also ensure that ground definitions are found whenever possible, we observe that a similar bottom up approach identifies terms that can be rewritten into ground ones. This builds on the notion of constructively ground nodes, defined next.

A class c is *ground* if c contains a *constructively ground*, or *c-ground* for short, node n, where a node n is c-ground if either (a) $\mathit{term}(n)$ is ground, or (b) n is not a leaf and the class $\mathit{class}(n[i])$ of every child $n[i]$ is ground. Note that nodes labeled by variables are never c-ground.

In the example in Fig. 1, $\mathit{class}(\mathsf{N}(7))$ and $\mathit{class}(\mathsf{N}(8))$ are not ground, because all their nodes represent variables; $\mathit{class}(\mathsf{N}(6))$ is ground because $\mathsf{N}(6)$ is c-ground. Nodes $\mathsf{N}(4)$ and $\mathsf{N}(5)$ are not c-ground because the class of $\mathsf{N}(8)$ (a child of both nodes) is not ground. Interestingly, $\mathsf{N}(1)$ is c-ground, because $\mathit{class}(\mathsf{N}(3)) = \mathit{class}(\mathsf{N}(6))$ is ground, even though its term $3 > z$ is not ground.

Ground classes and c-ground nodes are of interest because whenever $\varphi \models \mathit{term}(n) \approx t$ for some node n and ground term t, then $\mathit{class}(n)$ is ground, i.e., it contains a c-ground node, where c-ground nodes can be found recursively starting from ground leaves. Furthermore, the recursive definition ensures that

when the aforementioned c-ground nodes are selected as representatives, the corresponding terms w.r.t. `repr` are ground.

As a result, to maximize the ground definitions found, we are interested in finding an admissible representative function `repr` that is *maximally ground*, which means that for every node $n \in N$, if $class(n)$ is ground, then $\texttt{repr}(n)$ is c-ground. That means that c-ground nodes are always chosen if they exist.

Theorem 2. *Let $G = egraph(\varphi)$ be an egraph and* `repr` *an admissible representative function that is maximally ground. For all $n \in N$, if $\varphi \models term(n) \approx t$ for some ground term t, then $\texttt{repr}(n)$ is c-ground and $\texttt{ntt}(\texttt{repr}(n))$ is ground.*

We note that not every choice of c-ground nodes as representatives results in an admissible representative function. For example, consider the formula φ_4 of Fig. 4 and its egraph. All nodes except for $\mathsf{N}(5)$ and $\mathsf{N}(2)$ are c-ground. However, a `repr` with $\mathsf{N}(3)$ and $\mathsf{N}(1)$ as representatives is not admissible. Intuitively, this is because the "witness" for c-groundness of $\mathsf{N}(1)$ in $class(\mathsf{N}(2))$ is $\mathsf{N}(4)$ and not $\mathsf{N}(3)$. Therefore, it is important to incorporate the selection of c-ground representatives into the bottom up procedure that ensures admissibility of `repr`.

To promote c-ground nodes over non c-ground in the construction of an admissible representative function, `find_defs` chooses representatives in two steps. First, only the ground leaves are processed (line 2). This ensures that c-ground representatives are chosen while guaranteeing the absence of cycles. Then, the remaining leaves are added to *todo* (line 4). This triggers representative selection of the remaining classes (those that are not ground).

We illustrate `find_defs` with two examples. For φ_4 of Fig. 4, there is only one leaf that is ground, $\mathsf{N}(4)$, which is added to *todo* on line 2, and *todo* is processed. $\mathsf{N}(4)$ is chosen as representative and, as a consequence, its parent $\mathsf{N}(1)$ is added to *todo*. $\mathsf{N}(1)$ is chosen as representative so $\mathsf{N}(3)$, even though added to the queue later, is not chosen as representative, obtaining $\texttt{repr}_{4b} = \{\mathsf{N}(4), \mathsf{N}(1)\}$. For φ_5 of Fig. 5, no nodes are added to *todo* on line 2. $\mathsf{N}(3)$ and $\mathsf{N}(6)$ are added on line 4. In `process`, both are chosen as representatives obtaining, $\texttt{repr}_{5b}$.

Alg. 2 guarantees that `repr` is maximally ground. Together with Theorem 2, this implies that all terms that can be rewritten into ground equivalent ones will be rewritten, which, in turn, means that for each variable that has a ground definition, its representative is one such definition.

Finding Additional (Non-ground) Definitions. At this point, QEL found ground definitions while avoiding cycles in $G_{\texttt{repr}}$. However, this does not mean that as many variables as possible are eliminated. A variable can also be eliminated if it can be expressed as a function of other variables. This is not achieved by `find_defs`. For example, in $\texttt{repr}_{5b}$ both variables are representatives, hence none is eliminated, even though, since $x \approx g(f(y))$, x could be eliminated in φ_5 by rewriting x as a function of y. Alg. 3 shows function `refine_defs` that refines maximally ground `repr`*s* to further find such definitions while keeping admissibility and ground maximality. This is done by greedily attempting to change class representatives if they are labeled with a variable. `refine_defs`

```
egraph :: refine_defs(repr, v)
 1: for n ∈ N do
 2:   if n = repr(n) and L(n) ∈ v then
 3:     r := n
 4:     for n' ∈ class(n) \ {n} do
 5:       if L(n') ∉ v then
 6:         if not cycle(n', repr) then
 7:           r := n';
 8:           break
 9:     for n' ∈ class(n) do
10:       repr[n'] := r
11: ret repr
```

```
egraph :: find_core(repr, v)
1: core := ∅
2: for n ∈ N s.t. n = repr(n) do
3:   core := core ∪ {n}
4:   for n' ∈ (class(n) \ n) do
5:     if L(n') ∈ v then continue
6:     else if ∃m ∈ core · m congruent with n'
         then
7:       continue
8:     core := core ∪ {n'}
9: ret core
```

Algorithm 3: Refining `repr` and building core.

iterates over the nodes in the class checking if there is a different node that is not a variable and that does not create a cycle in $G_{\mathtt{repr}}$ (line 6). The resulting `repr` remains maximally ground because representatives of ground classes are not changed.

For example, let us refine $\mathtt{repr}_{5b} = \{\mathsf{N}(3), \mathsf{N}(6), \mathsf{N}(5)\}$ obtained for φ_5. Assume that x is processed first. For $class(\mathsf{N}(x))$, changing the representative to $\mathsf{N}(1)$ does not introduce a cycle (see Fig. 5c), so $\mathsf{N}(1)$ is selected. Next, for $class(\mathsf{N}(y))$, choosing $\mathsf{N}(4)$ causes $G_{\mathtt{repr}}$ to be cyclic since $\mathsf{N}(1)$ was already chosen (Fig. 5a), so the representative of $class(\mathsf{N}(y))$ is not changed. The final refinement is $\mathtt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$.

At this point, QEL found a representative function `repr` with as many ground definitions as possible and attempted to refine `repr` to have fewer variables as representatives. Next, QEL finds a core of the nodes of the egraph, based on `repr`, that will govern the translation of the egraph to a formula. While `repr` determines the semantic rewrites of terms that enable variable elimination, it is the use of the core in the translation that actually eliminates them.

Variable Elimination Based on a Core. A *core* of an egraph $G = \langle N, E, L, \mathtt{root}\rangle$ and a representative function `repr`, is a subset of the nodes $N_c \subseteq N$ such that $\psi_c = G.\mathtt{to_formula}(\mathtt{repr}, N \setminus N_c)$ satisfies $isFormula(G, \psi_c)$.

Alg. 3 shows pseudocode for `find_core` that computes a core of an egraph for a given representative function. The idea is that non-representative nodes that are labeled by variables, as well as nodes congruent to nodes that are already in the core, need not be included in the core. The former are not needed since we are only interested in preserving the existential closure of the output, while the latter are not needed since congruent nodes introduce the same syntactic terms in the output. For example, for φ_1 and $\mathtt{repr}_1$, `find_core` returns $\mathtt{core}_1 = N_1 \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. Nodes $\mathsf{N}(3)$ and $\mathsf{N}(9)$ are excluded because they are labeled with variables; and node $\mathsf{N}(5)$ because it is congruent with $\mathsf{N}(4)$.

Finally, QEL produces a quantifier reduction by applying `to_formula` with the computed `repr` and `core`. Variables that are not in the core (they are not representatives) are eliminated – this includes variables that have a ground defi-

nition. However, QEL may eliminate a variable even if it is a representative (and thus it is in the core). As an example, consider $\psi(x, y) \triangleq f(x) \approx f(y) \land x \approx y$, whose egraph G contains 2 classes with 2 nodes each. The core N_c relative to any admissible $\texttt{repr}$ contains only one representative per class: in the $\mathit{class}(\mathsf{N}(x))$ because both nodes are labeled with variables, and in the $\mathit{class}(\mathsf{N}(f(x)))$ because nodes are congruent. In this case, $\texttt{to_formula}(\texttt{repr}, N_c)$ results in $\top$ (since singleton classes in the core produce no literals in the output formula), a quantifier elimination of ψ. More generally, the variables are eliminated because none of them is reachable in $G_{\texttt{repr}}$ from a non-singleton class in the core (only such classes contribute literals to the output).

We conclude the presentation of QEL by showing its output for our examples. For φ_1, QEL obtains $(k + 1 \approx \mathit{read}(a, x) \land 3 > k + 1)$, a quantifier reduction, using $\texttt{repr}_1 = \{\mathsf{N}(3), \mathsf{N}(8))\}$ and $\texttt{core}_1 = N_1 \setminus \{\mathsf{N}(3), \mathsf{N}(5), \mathsf{N}(9)\}$. For φ_4, QEL obtains $(6 \approx f(g(6)))$, a quantifier elimination, using $\texttt{repr}_{4b} = \{\mathsf{N}(4), \mathsf{N}(1)\}$, and $\texttt{core}_{4b} = N_4 \setminus \{\mathsf{N}(3), \mathsf{N}(2)\}$. Finally, for φ_5, QEL obtains $(y \approx h(f(y)) \land f(g(f(y))) \approx f(y))$, a quantifier reduction, using $\texttt{repr}_{5c} = \{\mathsf{N}(1), \mathsf{N}(6), \mathsf{N}(5)\}$ and $\texttt{core}_{5c} = N_5 \setminus \{\mathsf{N}(3)\}$.

Guarantees of QEL. Correctness of QEL is straightforward. We conclude this section by providing two conditions that ensure that a variable is eliminated by QEL. The first condition guarantees that a variable is eliminated whenever a ground definition for it exists (regardless of the specific representative function and core computed by QEL). This makes QEL *complete relative to quantifier elimination based on ground definitions*. Relative completeness is an important property since it means that QEL is unaffected by variable orderings and syntactic rewrites, unlike QELite. The second condition, illustrated by ψ above, depends on the specific representative function and core computed by QEL.

Theorem 3. *Let φ be a QF conjunction of literals with free variables $\boldsymbol{v}$, and let $v \in \boldsymbol{v}$. Let $G = \mathit{egraph}(\varphi)$, n_v the node in G such that $L(n_v) = v$ and* $\texttt{repr}$ *and* $\texttt{core}$ *computed by QEL. We denote by $NS = \{n \in \texttt{core} \mid (\mathit{class}(n) \cap \texttt{core}) \neq \{n\}\}$ the set of nodes from classes with two or more nodes in* $\texttt{core}$. *If one of the following conditions hold, then v does not appear in QEL$(\varphi, \boldsymbol{v})$:*

(1) there exists a ground term t s.t. $\varphi \models v \approx t$, or
(2) n_v is not reachable from any node in NS in $G_{\texttt{repr}}$.

As a corollary, if every variable meets one of the two conditions, then QEL finds a quantifier elimination.

This concludes the presentation of our quantifier reduction algorithm. Next, we show how QEL can be used to under-approximate quantifier elimination, which allows working with formulas for which QEL does not result in a qelim.

5 Model Based Projection Using QEL

Applications like model checking and quantified satisfiability require efficient computation of under-approximations of quantifier elimination. They make use

$$\textsc{ElimWrRd1}\quad \frac{\varphi[read(write(t,i,v),j)]}{\varphi[v] \land i \approx j}\; M \models i \approx j$$

$$\textsc{ElimWrRd2}\quad \frac{\varphi[read(write(t,i,v),j)]}{\varphi[read(t,j)] \land i \not\approx j}\; M \models i \not\approx j$$

Fig. 6: Two MBP rules from [16]. The notation $\varphi[t]$ means that φ contains term t. The rules rewrite all occurrences of $read(write(t, i, v), j)$ with v and $read(t, j)$, respectively.

ElimWrRd

```
1: function match(t)
2:   ret t = read(write(s, i, v), j)
3: function apply(t, M, G)
4:   if M ⊨ i ≈ j then
5:     G.assert(i ≈ j)
6:     G.assert(t ≈ v)
7:   else
8:     G.assert(i ≉ j)
9:     G.assert(t ≈ read(s, j))
```

Fig. 7: Adaptation of rules in Fig. 6 using QEL API.

of model-based projection (MBP) algorithms to project variables that cannot be eliminated cheaply. Our QEL algorithm is efficient and relatively complete, but it does not guarantee to eliminate all variables. In this section, we use a model and theory-specific projection rules to implement an MBP algorithm on top of QEL.

We focus on two important theories: Arrays and Algebraic DataTypes (ADT). They are widely used to encode program verification tasks. Prior works separately develop MBP algorithms for Arrays [16] and ADTs [5]. Both MBPs were presented as a set of syntactic rewrite rules applied until fixed point.

Combining the MBP algorithms for Arrays and ADTs is non-trivial because applying projection rules for one theory may produce terms of the other theory. Therefore, separately achieving saturation in either theory is not sufficient to reach saturation in the combined setting. The MBP for the combined setting has to call both MBPs, check whether either one of them produced terms that can be processed by the other, and, if so, call the other algorithm. This is similar to theory combination in SMT solving where the core SMT solver has to keep track of different theory solvers and exchange terms between them.

Our main insight is that egraphs can be used as a glue to combine MBP algorithms for different theories, just like egraphs are used in SMT solvers to combine satisfiability checking for different theories. Implementing MBP using egraphs allows us to use the insights from QEL to combine MBP with on-the-fly quantifier reduction to produce less under-approximate formulas than what we get by syntactic application of MBP rules.

To implement MBP using egraphs, we implement all rewrite rules for MBP in Arrays [16] and ADTs [5] on top of egraphs. In the interest of space, we explain the implementation of just a couple of the MBP rules for Arrays[6].

Fig. 6 shows two Array MBP rules from [16]: ElimWrRd1 and ElimWrRd2. Here, φ is a formula with arrays and M is a model for φ. Both rules rewrite terms which match the pattern $read(write(t, i, v), j)$, where t, i, j, k are all terms and t contains a variable to be projected. ElimWrRd1 is applicable when $M \models i \approx j$. It rewrites the term $read(write(t, i, v), j)$ to v. ElimWrRd2 is applicable when $M \not\models i \approx j$ and rewrites $read(write(t, i, v), j)$ to $read(t, j)$.

[6] Implementation of all other rules is similar.

Fig. 7 shows the egraph implementation of ElimWrRd1 and ElimWrRd2. The `match`(t) method checks if t syntactically matches $read(write(s,i,v),j)$, where s contains a variable to be projected. The `apply`(t) method assumes that t is $read(write(s,i,v),j)$. It first checks if $M \models i \approx j$, and, if so, it adds $i \approx j$ and $t \approx v$ to the egraph G. Otherwise, if $M \not\models i \approx j$, `apply`($t$) adds a disequality $i \not\approx j$ and an equality $t \approx read(s,v)$ to G. That is, the egraph implementation of the rules only adds (and does not remove) literals that capture the side condition and the conclusion of the rule.

Our algorithm for MBP based on egraphs, MBP-QEL, is shown in Alg. 4. It initializes an egraph with the input formula (line 1), applies MBP rules until saturation (line 4), and then uses the steps of QEL (lines 7–12) to generate the projected formula.

Applying rules is as straightforward as iterating over all terms t in the egraph, and for each rule r such that r.`match`(t) is true, calling r.`apply`(t, M, G) (lines 14–22). As opposed to the standard approach based on formula rewriting, here the terms are *not* rewritten – both remain. Therefore, it is possible to get into an infinite loop by re-applying the same rules on the same terms over and over again. To avoid this, MBP-QEL marks terms as *seen* (line 23) and avoids them in the next iteration (line 15). Some rules in MBP are applied to pairs of terms. For example, Ackermann rewrites pairs of *read* terms over the same variable. This is different from usual applications where rewrite rules are applied to individual expressions. Yet, it is easy to adapt such pairwise rewrite rules to egraphs by iterating over pairs of terms (lines 25–30).

MBP-QEL does not apply MBP rules to terms that contain variables but are already c-ground (line 16), which is sound because such terms are replaced by ground terms in the output (Theorem 3). This prevents unnecessary application of MBP rules thus allowing MBP-QEL to compute MBPs that are closer to a quantifier elimination (less model-specific).

Just like each application of a rewrite rule introduces a new term to a formula, each call to the `apply` method of a rule adds new terms to the egraph. Therefore, each call to *ApplyRules* (line 4) makes the egraph bigger. However, provided that the original MBP combination is terminating, the iterative application of *ApplyRules* terminates as well (due to marking).

Some MBP rules introduce new variables to the formula. MBP-QEL computes `repr` based on both original and newly introduced variables (line 7). This allows MBP-QEL to eliminate all variables, including non-Array, non-ADT variables, that are equivalent to ground terms (Theorem 3).

As mentioned earlier, MBP-QEL never removes terms while rewrite rules are saturating. Therefore, after saturation, the egraph still contains all original terms and variables. From soundness of the MBP rules, it follows that after each invocation of `apply`, MBP-QEL creates an under-approximation of $\varphi^{\exists}$ based on the model M. From completeness of MBP rules, it follows that, after saturation, all terms containing Array or ADT variables can be removed from the egraph without affecting equivalence of the saturated egraph. Hence, when calling `to_formula`, MBP-QEL removes all terms containing Array or ADT

Input: A QF formula φ with free variables $\boldsymbol{v}$ all of sort $Array(I, V)$ or ADT, a model $M \models \varphi^\exists$, and sets of rules $ArrayRules$ and $ADTRules$
Output: A cube ψ s.t. $\psi^\exists \Rightarrow \varphi^\exists$, $M \models \psi^\exists$, and $vars(\psi)$ are not Arrays or ADTs

MBP-QEL($\varphi, \boldsymbol{v}, M$)

1: $G := egraph(\varphi)$
2: $p_1, p_2 := \top, \top$; $S, S_p := \emptyset, \emptyset$
3: **while** $p_1 \vee p_2$ **do**
4: $\quad p_1 := ApplyRules(G, M, ArrayRules, S, S_p)$
5: $\quad p_2 := ApplyRules(G, M, ADTRules, S, S_p)$
6: $\boldsymbol{v}' := G.Vars()$
7: $\texttt{repr} := G.\texttt{find_defs}(\boldsymbol{v}')$
8: $\texttt{repr} := G.\texttt{refine_defs}(\texttt{repr}, \boldsymbol{v}')$
9: $\texttt{core} := G.\texttt{find_core}(\texttt{repr}, \boldsymbol{v}')$
10: $\boldsymbol{v}_e := \{v \in \boldsymbol{v}' \mid is_arr(v) \vee is_adt(v)\}$
11: $\texttt{core}_e := \{n \in \texttt{core} \mid gr(term(n), \boldsymbol{v}_e)\}$
12: **ret** $G.\texttt{to_formula}(\texttt{repr}, G.Nodes() \setminus \texttt{core}_e)$

$ApplyRules(G, M, R, S, S_p)$

13: $progress := \bot$
14: $N := G.Nodes()$
15: $U := \{n \mid n \in N \setminus S\}$
16: $T := \{term(n) \mid n \in U \wedge (is_eq(term(n)) \vee \neg c\text{-}ground(n))\}$
17: $R_p := \{r \in R \mid r.is_for_pairs()\}$
18: $R_u := R \setminus R_p$
19: **for each** $t \in T, r \in R_u$ **do**
20: $\quad$ **if** $r.\texttt{match}(t)$ **then**
21: $\quad\quad r.\texttt{apply}(t, M, G)$
22: $\quad\quad progress := \top$
23: $S := S \cup N$
24: $N_p := \{\langle n_1, n_2 \rangle \mid n_1, n_2 \in N\}$
25: $T_p := \{term(n_p) \mid n_p \in N_p \setminus S_p\}$
26: **for each** $t_p \in T_p, r \in R_p$ **do**
27: $\quad$ **if** $r.\texttt{match}(p)$ **then**
28: $\quad\quad r.\texttt{apply}(p, M, G)$
29: $\quad\quad progress := \top$
30: $S_p := S_p \cup N_p$
31: **ret** $progress$

Algorithm 4: MBP-QEL: an MBP using QEL. Here $gr(t, \boldsymbol{v})$ checks whether term t contains any variables in $\boldsymbol{v}$ and $is_eq(t)$ checks if t is an equality literal.

variables (line 12). This includes, in particular, all the terms on which rewrite rules were applied, but potentially more.

We demonstrate our MBP algorithm on an example with nested ADTs and Arrays. Let $P \triangleq \langle A_{I \times I}, I \rangle$ be the datatype of a pair of an integer array and an integer, and let $pair : A_{I \times I} \times I \to P$ be its sole constructor with destructors $fst : P \to A_{I \times I}$ and $snd : P \to I$. In the following, let i, l, j be integers, a an integer array, p, p' pairs, and $\boldsymbol{p}_1, \boldsymbol{p}_2$ arrays of pairs ($A_{I \times P}$). Consider the formula:

$$\varphi_{mbp}(p, a) \triangleq read(a, i) \approx i \wedge p \approx pair(a, l) \wedge \boldsymbol{p}_2 \approx write(\boldsymbol{p}_1, j, p) \wedge p \not\approx p'$$

where p and a are free variables that we want to project and all of $i, j, l, \boldsymbol{p}_1, \boldsymbol{p}_2, p'$ are constants that we want to keep. MBP is guided by a model $M_{mbp} \models \varphi_{mbp}$. To eliminate p and a, MBP-QEL constructs the egraph of φ_{mbp} and applies the MBP rules. In particular, it uses Array MBP rules to rewrite the $write(\boldsymbol{p}_1, j, p)$ term by adding the equality $read(\boldsymbol{p}_2, j) \approx p$ and merging it with the equivalence class of $\boldsymbol{p}_2 \approx write(\boldsymbol{p}_1, j, p)$. It then applies ADT MBP rules to deconstruct the equality $p \approx pair(a, l)$ by creating two equalities $fst(p) \approx a$ and $snd(p) \approx l$. Finally, the call to `to_formula` produces

$$\begin{aligned}&\mathit{read}(\mathit{fst}(\mathit{read}(\boldsymbol{p}_1, j)), i) \approx i \wedge \mathit{snd}(\mathit{read}(\boldsymbol{p}_1, j)) \approx l \wedge \\ &\qquad\qquad \mathit{read}(\boldsymbol{p}_2, j) \approx \mathit{pair}(\mathit{fst}(\mathit{read}(\boldsymbol{p}_1, j)), l) \wedge \\ &\qquad\qquad\qquad \boldsymbol{p}_2 \approx \mathit{write}(\boldsymbol{p}_1, j, \mathit{read}(\boldsymbol{p}_2, j)) \wedge \mathit{read}(\boldsymbol{p}_2, j) \not\approx p'\end{aligned}$$

The output is easy to understand by tracing it back to the input. For example, the first literal is a rewrite of the literal $\mathit{read}(a, i) \approx i$ where a is represented with $\mathit{fst}(p)$ and p is represented with $\mathit{read}(\boldsymbol{p}_1, j)$. While the interaction of these rules might seem straightforward in this example, the MBP implementation in Z3 fails to project a in this example because of the multilevel nesting.

Notably, in this example, the c-ground computation during projection allows MBP-QEL not splitting on the disequality $p \not\approx p'$ based on the model. While ADT MBP rules eliminate disequalities by using the model to split them, MBP-QEL benefits from the fact that, after the application of Array MBP rules, the class of p becomes ground, making $p \not\approx p'$ c-ground. Thus, the c-ground computation allows MBP-QEL to produce a formula that is less approximate than those produced by syntactic application of MBP rules. In fact, in this example, a quantifier elimination is obtained (the model M_{mbp} was not used).

In the next section, we show that our improvements to MBP translate to significant improvements in a CHC-solving procedure that relies on MBP.

6 Evaluation

We implement QEL (Alg. 1) and MBP-QEL (Alg. 4) inside Z3 [19] (version 4.12.0), a state-of-the-art SMT solver. Our implementation (referred to as Z3EG), is publicly available on GitHub[7]. Z3EG replaces QELITE with QEL, and the existing MBP with MBP-QEL.

We evaluate Z3EG using two solving tasks. Our first evaluation is on the QSAT algorithm [5] for checking satisfiability of formulas with alternating quantifiers. In QSAT, Z3 uses both QELITE and MBP to under-approximate quantified formulas. We compare three QSAT implementations: the existing version in Z3 with the default QELITE and MBP; the existing version in Z3 in which QELITE and MBP are replaced by our egraph-based algorithms, Z3EG; and the QSAT implementation in YICESQS[8], based on the YICES [8] SMT solver. During the evaluation, we found a bug in QSAT implementation of Z3 and fixed it[9]. The fix resulted in Z3 solving over 40 sat instances and over 120 unsat instances more than before. In the following, we use the fixed version of Z3.

We use benchmarks in the theory of (quantified) LIA and LRA from SMT-LIB [2,3], with alternating quantifiers. LIA and LRA are the only tracks in which Z3 uses the QSAT tactic by default. To make our experiments more comprehensive, we also consider two modified variants of the LIA and LRA benchmarks, where we add some non-recursive ADT variables to the benchmarks. Specifically, we wrap all existentially quantified arithmetic variables using a record

[7] Available at https://github.com/igcontreras/z3/tree/qel-cav23.
[8] Available at https://github.com/disteph/yicesQS.
[9] Available at https://github.com/igcontreras/z3/commit/133c9e438ce.

Cat.	Count	Z3EG		Z3		YicesQS	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
LIA	416	150	266	150	266	107	102
LRA	2 419	795	1 589	793	1 595	808	1 610

Table 1: Instances solved within 20 minutes by different implementations. Benchmarks are quantified **LIA** and **LRA** formulas from SMT-LIB [2].

Cat.	Count	Z3EG		Z3	
		SAT	UNSAT	SAT	UNSAT
LIA-ADT	416	150	266	150	56
LRA-ADT	2 419	757	1 415	196	964

Table 2: Instances solved within 60 seconds for our handcrafted benchmarks.

type ADT and unwrap them whenever they get used[10]. Since these benchmarks are similar to the original, we force Z3 to use the QSAT tactic on them with a `tactic.default_tactic=qsat` command line option.

Tab. 1 summarizes the results for the SMT-LIB benchmarks. In LIA, both Z3EG and Z3 solve all benchmarks in under a minute, while YicesQS is unable to solve many instances. In LRA, YicesQS solves all instances with very good performance. Z3 is able to solve only some benchmarks, and our Z3EG performs similarly to Z3. We found that in the LRA benchmarks, the new algorithms in Z3EG are not being used since there are not many equalities in the formula, and no equalities are inferred during the run of QSAT. Thus, any differences between Z3 and Z3EG are due to inherent randomness of the solving process.

Tab. 2 summarizes the results for the categories of mixed ADT and arithmetic. YicesQS is not able to compete because it does not support ADTs. As expected, Z3EG solves many more instances than Z3.

The second part of our evaluation shows the efficacy of MBP-QEL for Arrays and ADTs (Alg. 4) in the context of CHC-solving. Z3 uses both QeLite and MBP inside the CHC-solver Spacer [17]. Therefore, we compare Z3 and Z3EG on CHC problems containing Arrays and ADTs. We use two sets of benchmarks to test out the efficacy of our MBP. The benchmarks in the first set were generated for verification of Solidity smart contracts [1] (we exclude benchmarks with non-linear arithmetic, they are not supported by Spacer). These benchmarks have a very complex structure that nests ADTs and Arrays. Specifically, they contain both ADTs of Arrays, as well as Arrays of ADTs. This makes them suitable to test our MBP-QEL. Row 1 of Tab. 3 shows the number of instances solved by Z3 (Spacer) with and without MBP-QEL. Z3EG solves 29 instances more than Z3. Even though MBP is just one part of the overall Spacer algorithm, we see that for these benchmarks, MBP-QEL makes a significant impact on Spacer. Digging deeper, we find that many of these instances come from the category called *abi* (row 2 in Tab. 3). Z3EG solves all of these benchmarks, while Z3 fails to solve 20 of them. We traced the problem down to the MBP implementation in Z3: it fails to eliminate all variables, causing runtime exception. In contrast, MBP-QEL eliminates all variables successfully, allowing Z3EG to solve these benchmarks.

[10] The modified benchmarks are available at https://github.com/igcontreras/LIA-ADT and https://github.com/igcontreras/LRA-ADT.

Cat.	Count	Z3EG		Z3		ELDARICA	
		SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
Solidity	3 468	2 324	1 133	2 314	1 114	**2 329**	**1 134**
↳ **abi**	127	19	108	19	88	19	108
LIA-lin-Arrays	488	**214**	72	212	**75**	147	68

Table 3: Instances solved within 20 minutes by Z3EG, Z3, and ELDARICA. Benchmarks are CHCs from **Solidity** [1] and CHC competition [13]. The **abi** benchmarks are a subset of **Solidity** benchmarks.

We also compare Z3EG with ELDARICA [14], a state-of-the-art CHC-solver that is particularly effective on these benchmarks. Z3EG solves almost as many instances as ELDARICA. Furthermore, like Z3, Z3EG is orders of magnitude faster than ELDARICA. Finally, we compare the performance of Z3EG on Array benchmarks from the CHC competition [13]. Z3EG is competitive with Z3, solving 2 additional safe instances and almost as many unsafe instances as Z3 (row 3 of Tab. 3). Both Z3EG and Z3 solve quite a few instances more than ELDARICA.

Our experiments show the effectiveness of our QEL and MBP-QEL in different settings inside the state-of-the-art SMT solver Z3. While we maintain performance on quantified arithmetic benchmarks, we improve Z3's QSAT algorithm on quantified benchmarks with ADTs. On verification tasks, QEL and MBP-QEL help SPACER solve 30 new instances, even though MBP is only a relatively small part of the overall SPACER algorithm.

7 Conclusion

Quantifier elimination, and its under-approximation, Model-Based Projection are used by many SMT-based decision procedures, including quantified SAT and Constrained Horn Clause solving. Traditionally, these are implemented by a series of syntactic rules, operating directly on the syntax of an input formula. In this paper, we argue that these procedures should be implemented directly on the egraph data-structure, already used by most SMT solvers. This results in algorithms that better handle implicit equality reasoning and result in easier to implement and faster procedures. We justify this argument by implementing quantifier reduction and MBP in Z3 using egraphs and show that the new implementation translates into significant improvements to the target decision procedures. Thus, our work provides both theoretical foundations for quantifier reduction and practical contributions to Z3 SMT-solver.

Acknowledgment The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 1810/18. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), MathWorks Inc., and the Microsoft Research PhD Fellowship.

References

1. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity Compiler's Model Checker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 325–338. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_16
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of model checking, pp. 305–343. Springer (2018)
5. Bjørner, N.S., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A. (eds.) 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015, Suva, Fiji, November 24-28, 2015. EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015). https://doi.org/10.29007/vv21
6. Chang, B.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3385, pp. 147–163. Springer (2005). https://doi.org/10.1007/978-3-540-30579-8_11
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (may 2005). https://doi.org/10.1145/1066100.1066102
8. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
9. Dutertre, B.: Solving Exists/Forall Problems with Yices. In: Workshop on Satisfiability Modulo Theories (2015), https://yices.csl.sri.com/papers/smt2015.pdf
10. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9583, pp. 85–103. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_4
11. Gascón, A., Subramanyan, P., Dutertre, B., Tiwari, A., Jovanovic, D., Malik, S.: Template-based circuit understanding. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. pp. 83–90. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987599
12. Gulwani, S., Tiwari, A., Necula, G.C.: Join algorithms for the theory of uninterpreted functions. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3328, pp. 311–323. Springer (2004). https://doi.org/10.1007/978-3-540-30538-5_26

13. Gurfinkel, A., Ruemmer, P., Fedyukovich, G., Champion, A.: CHC-COMP. `https://chc-comp.github.io/` (2018)
14. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–7. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
15. Joshi, R., Nelson, G., Randall, K.H.: Denali: A goal-directed superoptimizer. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002. pp. 304–314. ACM (2002). https://doi.org/10.1145/512529.512566
16. Komuravelli, A., Bjørner, N.S., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 89–96. IEEE (2015). https://doi.org/10.5555/2893529.2893548
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 17–34. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_2
18. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010. pp. 316–329. ACM (2010). https://doi.org/10.1145/1806596.1806632
19. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Nelson, G., Oppen, D.C.: Fast decision algorithms based on union and find. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 114–119. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.12
21. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979). https://doi.org/10.1145/357073.357079
22. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 264–276. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480915
23. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. Log. Methods Comput. Sci. **7**(1) (2011). https://doi.org/10.2168/LMCS-7(1:10)2011

24. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304