Towards a Relational E-graph

Yihong Zhang

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2022

Reading Committee:
Professor Zachary Tatlock

Computer Science & Engineering

University of Washington

Abstract

Towards a Relational E-graph

Yihong Zhang

This thesis presents my experience improving the performance and expressiveness of e-graphs with both relational and non-relational approaches. Chapter 1 presents a non-relational optimization algorithm for e-matching and studies its properties. Motivated by the insufficiency of the non-relational approach, we turn to study ways to encode e-graphs inside Datalog in Chapter 2. We also describe a prototype language, named egglog, for relational e-graphs and its semantics in Chapter 3. Chapter 4 gives a survey of related work, and Chapter 5 concludes.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This thesis summarizes the work done during my master on building and understanding relational e-graphs. It builds upon my previous work on relational e-matching during the undergrad. Both my master and undergrad are done at UW, and after my master's journey, I will continue to work as a PhD student, also at UW, on the topic presented in this thesis.

I want to thank my advisor Professor Zachary Tatlock for his guidance during the development of this thesis and for his mentorship over the past years. I would also like to thank my collaborators Max Willsey, Remy Wang, Philip Zucker, and Eli Rosenthal for many insightful discussions. I also want to thank Professor Dan Suciu for all the insightful comments. Finally, I want to thank my family and friends for their support.

DEDICATION

To Chinese Feminist Activists.

To Huang Xueqin, Xianzi, and many others.

Chapter 1

OPTIMIZING NON-RELATIONAL E-MATCHING

E-matching is an important procedure for many e-graph based applications, yet it is slow. In a typical application of equality saturation, e-matching can take 60-90% of the overall run time (Willsey et al., 2021). In the work presented in my bachelor's thesis, my collaborators and I proposed a relational approach to e-matching, dubbed relational e-matching (Zhang et al., 2022; Zhang, 2021). In particular, we made e-matching orders of magnitude faster, proved theoretical bounds of e-matching, and opened the door for all kinds of wild optimizations that can be done with databases and e-graphs.

However, the relational e-matching approach also has some secret pitfalls. In particular, to have the best of both efficient e-graph maintenance and efficient e-matching, one has to switch back and forth between the e-graph to its relational representation. Our prototype[1] builds a relational database and associated indices from scratch for each match-apply iteration. This is acceptable in the equality saturation setting. E-matching and updates always alternate in batches, so the cost of building the database is amortized. Plus, since building databases and indices are both linear time costs, they are often subsumed by the time spent on e-matching.

However, what if e-matching is not run in batches? Or what if all the e-matching patterns are quite simple and the constant overhead is now a bottleneck? An e-graph framework can implement some fast paths for that, but then there are more design questions to consider: Are we going to keep two implementations of e-matching? What kind of queries should be computed by relational e-matching and what by traditional e-matching? ... We can continue down this path and put a lot of engineering effort into building a practically efficient e-graph

[1] https://github.com/egraphs-good/egg/tree/relational

engine, or we can:

1. Start a clean-slate relational e-graph framework that handles all e-graph operations efficiently and forget about the graph part of an e-graph;

2. Keep the current e-graph data structure, and port some optimizations of relational e-matching back to e-graphs.

In this chapter, I will focus on the second approach. The rest of the thesis will focus on the first approach. When working on relational e-matching, we found an optimization to the backtracking-style classical e-matching. Like relational e-matching, it is able to improve e-matching asymptotically in some cases, but it does not require transforming the input e-graph to a relational database. And it is very simple. For what it is worth, this optimization (instead of relational e-matching) is what is currently implemented in egg.

In this chapter, I will describe this optimization. But before that, we will go over some brief background on e-matching. Readers are welcome to skip it if they already know what e-matching is.

1.1 E-Matching

There have been many great introductions to e-graphs and e-matching. For example, Philip Zucker gives a gentle introduction to e-graphs (Zucker, 2020b) and e-matching (Zucker, 2020a) in Julia. Max Willsey also wrote a very nice tutorial (Willsey, 2020) on e-graphs and egg (Willsey et al., 2021). Basically, an e-graph is a data structure that compactly represents an equivalence relation and e-matching is pattern matching on such e-graphs modulo equivalence. Both e-matching and e-graphs are widely used in SMT solvers (de Moura and Bjørner, 2007) and equality saturation-based program optimizers (Yang et al., 2021). A typical equality saturation-based program optimizer may take the majority of its time doing e-matching.

There are several algorithms proposed for e-matching. For example, the one currently used in egg is based on the virtual machine proposed by de Moura and Bjørner (2007). The traditional backtracking-based e-matching algorithm does not exploit equality constraints during pattern compilation. Equality constraints are the term we used in the relational e-matching paper to describe the kind of constraints that all occurrences of the same variables should be mapped to equivalent terms. Those that violate the equality constraints will not be pruned away immediately. For example, $f(\alpha, g(\alpha))$ does not match $f(1, g(2))$, because the first α is mapped to 1 but the second is mapped to 2. The classical backtracking-based e-matching will still consider it though.

The relational e-matching approach instead treats an e-matching pattern as a kind of relational query. From a relational query, the query optimizer can easily identify all kinds of constraints, including equality constraints, and find an efficient query plan. As an example, the above pattern can be compiled to query $Q(r, \alpha) \leftarrow R_f(r, \alpha, x), R_g(x, \alpha)$, and a hash join could answer this query in linear time.

1.2 The optimization

The issue with the traditional backtracking-style e-matching is that it does not take advantage of the equality constraints, so it enumerates obviously unsatisfying terms. The optimization is therefore straightforward: do not enumerate terms that are obviously unsatisfying. And this is easy, because we already know what the (only) satisfying term should look like!

Let us first look at the classical e-matching algorithm (reproduced from Zhang et al. (2022, Figure 3), with a typo fix).

$$match(x, c, S) = \{\sigma \cup \{x \mapsto c\} \mid \sigma \in S, x \notin \mathrm{dom}(\sigma)\} \cup \{\sigma \mid \sigma \in S, \sigma(x) = c\}$$

$$match(f(p_1, \ldots, p_k), c, S) = \bigcup_{f(c_1,\ldots,c_k) \in c} match(p_k, c_k, \ldots, match(p_1, c_1, S))$$

It takes a pattern p, an e-class c, and current substitutions S, and returns the set of substitutions produced by e-matching p over e-class c, such that all produced substitutions are extensions of some substitutions in S. The result of e-matching a pattern p over an e-graph is $\bigcup_{c \in C} match(p, c, \{\emptyset\})$ (both Zhang et al. (2022) and de Moura and Bjørner (2007) have another typo here), where C is the set of e-classes in the e-graph.

The algorithm is straightforward:

1. If the pattern is a variable, and
   (a) if this variable is fresh in the domain of the substitution, then it is safe to extend the substitutions with $\{x \mapsto c\}$, or
   (b) if this variable is not fresh, we keep only these substitutions that are consistent with the mapping $\{x \mapsto c\}$.
2. If the pattern is a function symbol of the form $f(p_1, \ldots, p_k)$, the algorithm iterates over f-nodes $f(c_1, \ldots, c_k)$ in the e-class, and fold over the sub patterns and sub e-classes with *match*, to accumulate set of valid substitutions.

The trick is to generalize case 1.b. In case 1.b, we know the substitution for a pattern is unique when the pattern is a non-fresh variable, but we *also* know this when the variables of the pattern are in the domain of the substitution (i.e., $\mathrm{fv}(p) \subseteq \mathrm{dom}(S)$), thanks to canonicalization. In that case, the pattern after substitution is a ground term, which can be efficiently looked up in a bottom-up fashion.

To implement this idea, we lift case 1.b to the top-level of the algorithm. During e-matching, the algorithm will first check whether the free vars of the input pattern is contained

in the domain of the substitution. If yes, then instead of looking further into the pattern, the algorithm will lookup the substituted term for comparison. The following definition shows this:

$$match(p, c, S) = \begin{cases} \{\sigma \mid \sigma \in S, lookup([\sigma]e) = c\} & \text{if } \mathrm{fv}(p) \subseteq \mathrm{dom}(S) \\ match'(p, c, S) & \text{o.w.} \end{cases}$$

$$match'(x, c, S) = \{\sigma \cup \{x \mapsto c\} \mid \sigma \in S\}$$

$$match'(f(p_1, \ldots, p_k), c, S) = \bigcup_{f(c_1,\ldots,c_k) \in c} match(p_k, c_k, \ldots, match(p_1, c_1, S))$$

In the above definition, we also drop the check of $x \notin \mathrm{dom}(\sigma)$ for the variable case, which is guaranteed not to happen.

As an example, consider $f(\alpha, g(\alpha))$ again. E-matching will enumerate through each f-node and bind α to the first child of the f-node. Here, the classical e-matching algorithm will then enumerate though the second child e-class of the f-node for possible g-nodes. However, because $g(\alpha)$ is a ground term after substituting α with $\sigma(\alpha)$, we can effectively lookup $g(\alpha)$ and compare it with the e-class id of the second child. The pseudocode:

```
# classical e-matching
for f in c: # f(a, g(a))
  for g in f.child2: # g(a)
    if f.child1 != g.child1:
      continue
    yield {a: f.child1}

# with the trick
for f in c: # f(a, g(a))
  g = lookup(mk_node(g, f.child1))
  if g is None or g != f.child2:
    continue
```

```
yield {a: f.child1}
```

Implementation-wise, egg adds a new operator to the e-matching virtual machine called `Lookup`. `Lookup` (1) substitutes the pattern with values in the VM register to produce a ground term and (2) lookup the ground term in the e-graph.

1.3 A Relational View of the Trick

How effective is this trick? To have a better understanding of this trick, we need to take a relational lens. The classical e-matching can be viewed as a relational query plan where hash joins only index one column (the link between parent and child) and potentially prune using the rest of equality columns (the equality constraint). At first I thought this optimization will make classical e-matching equivalent to some efficient hash join-based query plans, and a efficient plan here specifically means a plan where the hash joins will index all the columns known to be equivalent. But this is false. Consider the pattern $f(\alpha, g(\alpha, \beta))$. The relational version of it is $Q(r, \alpha, \beta) \leftarrow R_f(r, \alpha, x), R_g(x, \alpha, \beta)$. An efficient plan with hash joins will index both α and x. However, our trick cannot use the α in f to prune the α in g, because there could be multiple satisfying g-nodes (due to the unbound variable β). In this case, our optimization does not offer any speedup.

In fact, this trick can be relationally thought of as the kind of query optimizations that leverage functional dependencies. In the relational representation of e-graphs, there is a functional dependency from the children columns to the id column. For example, in relation $R_f(x, c_1, c_2)$, the relational representation of binary function symbol f, every combination of c_1 and c_2 uniquely determines x thanks to e-graph canonicalization. Our trick uses this information to immediately determine the value of x once c_1 and c_2 are known, without looking at obviously unsatisfying candidates.

In the relational e-matching paper, we also described how we use functional dependency to speed up queries. In fact, if the variable ordering of generic joins follows the topological order of the (acyclic) functional dependency, the run-time complexity will be worst-case optimal

under the presence of FDs (Ngo, 2018), a stronger guarantee than the original AGM bound (Atserias et al., 2008). Functional dependencies are also exploited for query optimization in databases (Kossmann et al., 2022).

How does this compare to relational e-matching? First, as we saw above, it is not as powerful as relational e-matching. Moreover, the graph representation has the fundamental restriction that makes it very hard to do advanced optimizations, e.g., one that uses cardinality information. It is also limited in the kind of join it is able to (conceptually) perform (only hash joins). However, it integrates well with an existing non-relational e-graph framework, which relational e-matching fails to achieve.

1.4 Query planning

This trick also poses a new question for classical e-matching planning: what visit order should one use? In the above definition of our algorithm, we assumed a depth-first style order of processing, but this is not necessary. For example, after enumerating the top-level f-node in pattern $f(g(\alpha), h(\alpha, \beta))$, it will be most efficient to enumerate the h-node and lookup $[\sigma]g(\alpha)$ later. If however we first enumerate $g(\alpha)$, we still cannot avoid enumerating $h(\alpha, \beta)$ later on.

If we assume the cost of enumerating each node is the same, this problem can be viewed as finding the smallest connected component (CC) in the pattern tree that contains the root, such that the CC covers all distinct variables. This is not an easy problem, and similar problems are NP-hard. This problem can be solved using dynamic programming on trees with exponential states, or can be reduced to an ILP problem. However, both seem to be an overkill for realistic queries.

It is also unclear what is a practically good planning heuristic. The one used in egg prioritizes sub-patterns with more free vars, but this may not be good enough: consider pattern $f(f(g(\alpha), \beta)), g(h(\alpha), h(\beta)))$. This heuristics yield the following plan for this pattern:

```
for f1 in c: # f(f(g(a), g(b))), g(h(a), h(b)))
```

```
for f2 in c.child1: # f(g(a), g(b))) (2 free vars)
  for g1 in c.child2: # g(h(a), h(b)) (2 free vars)
    for g2 in f2.child1: # g(a) (1 free var)
      h1 = lookup(mk_node(h, g2.child1) # lookup h(a)
      if h1 is None or h1 != g1.child1:
        continue
      for g3 in f2.child2: # g(b) (1 free var)
        h2 = lookup(mk_node(h, g3.child1) # lookup h(b)
        if h2 is None or h2 != g1.child2:
          continue
        yield {...}
```

This is complicated, but it suffices to only look at the first three loops: It does a cross product over the first and the second child of the top-level f-node. A good strategy here is instead to prefer fewer free vars, and performs the search in a depth-first search, so that $g(h(a), h(b))$ can be looked up all at once after $f(g(a), g(b))$ is enumerated. But it is not yet known if preferring fewer free vars is the right strategy. Moreover, realistic patterns tend to be small and simple, so cases like the above may be rare.

1.5 Miscellaneous

This chapter is adapted from my blog post *A Trick that Makes Classical E-Matching Faster* (Zhang, 2022). I thank Max and Phil for their valuable discussions and comments. The presented trick stems from a Pull Request[2] that tries to improve e-matching for ground terms. In hindsight, a variant of the proposed improvement targeting multi-patterns had been discussed in de Moura and Bjørner (2007) but was lost in egg's original e-matching implementation. Compared to that Pull Request, which only looks up ground terms, this optimization generalizes it by also looking up terms that are grounded after substitution.

[2]https://github.com/egraphs-good/egg/pull/74

Phil came up with this idea independently as well[3].

[3]https://github.com/egraphs-good/egg/pull/74#issuecomment-818833367

Chapter 2

ENCODING E-GRAPHS IN EXISTING DATALOG SYSTEMS

In Chapter 1, we discussed optimizations to make classical e-matching faster. As we see, there are still many limitations to the classical e-matching algorithm despite the proposed optimizations. Query plans are limited to certain special forms, so many queries are asymptotically slower using classical e-matching. Moreover, many advanced join algorithms (like the generic join algorithm) and optimizations (like ones using cardinality estimation) cannot be used due to the fundamental restriction of its graph representation. To enjoy the highly efficient e-matching procedure and the provided theoretical guarantees, we have to look back at the relational e-matching approach. However, relational e-matching has the "dual representation" problem: While e-matching is performed on the relaitonal representation, the graph representation is necessarily for standard e-graph operations like congruence maintenance. Therefore, both representations are needed and should be kept in sync for relational e-matching to work. This can have nontrivial overheads (Zhang et al., 2022).

A natural question to ask is, if keeping both representation is expensive, and efficient e-matching requires a relational representation, can we keep only the relational representation? This way, we are doing not only e-matching relationally, but also all other e-graph operations, and the ultimate goal is to be able to run equality saturation in this relational representation. Compared to the optimizations described in Chapter 1, this proposal is more radical, as it challenges the well-accepted assumption that an e-*egraph* is a graph. To implement this proposal, two key issues need to be addressed. First, equality saturation uses equational rewrites to grow the e-graph, so it is important to understand the semantics of rewrites in the relational representation. Second, a key ingredient to e-graphs is the maintenance of its congruence invariant. Therefore, a relational e-graph must be able to perform

congruence maintenance as well. To address the first issue, we propose to encode e-graphs in Datalog. Datalog is a relational language with rigourous semantics and efficient evaluation algorithms, where logic rules describe dependencies between relations. Logic rules in Datalog have the form $R(\ldots) :- R1(\ldots), \ldots, Rn(\ldots)$ and operationally performs fixpoint-based rewrites but for relations. Moreover, both rewrites in e-graphs and logics rules in Datalog are non-destructive, meaning that they do not remove original facts during the rewrites. Therefore, it is tempting to encode e-graph rewrites in Datalog.

This chapter introduces my experience encoding e-graph rewrites in two Datalog systems, namely Soufflé (Jordan et al., 2016) and Rel (Rel documentation team, 2022). Soufflé and Rel are different in many aspects, with different targeted use cases: Soufflé focuses on applications like program analyses and has a semantics very similar to the original Datalog, with "mild" extensions like algebraic data types (ADTs), built-in support for equivalence relations, and the choice operator. One of the most aggressive extension is perhaps the newly added subsumption, which allows the users to delete tuples when it is clear that they are subsumed by other more general tuples (Köstler et al., 1995). We will see subsumption is in fact critical in preventing the encoded e-graphs from blowup. Rel, in contrast, has more ambitious goals. While spiritually inspired by Datalog, Rel has a much more expressive front end language based on first-order logic. As an example, queries in Rel support universal quantifiers and existential quantifiers in arbitrary positions (as long as the domain of the quantified variables are finitely enumerable).

One important distinction between Soufflé and Rel is that Rel supports recursive aggregates out of box. Rigourous theories are developed for sound programming with recursive aggregates in Rel (Abo Khamis et al., 2022a), yet to facilitate even more flexible general-purpose programming, soundness are not enforced in practice. As a result, one needs to be careful when using recursive aggregates in Rel, to not violate properties like monotonicity. I use recursive aggregates in both encodings: while Rel supports it out of box, for Soufflé, I explicitly disabled the stratification checker. Despite the wide use of recursive aggregates, the encoding is still sound, because it is semantically clear that rewrites in an e-graph is

monotonic. Moreover, in the encoding, tuples are only removed when they are subsumed by a more canonicalized version of them, which intuitively provides a justification for the soundness.

A key ingredient to making e-graph efficient is to keep only the canonical tuples. However, the encoding in both systems are not completely satisfying. For Soufflé with the subsumption extension, a tuple can only be removed when it is able to find an evidence that this tuple is subsumed. For Rel, every iteration starts from scratch, so the only way to remove tuples is to recompute all the facts in the current iteration while excluding the removed tuples, which is indirect. Despite demonostrating the feasibility of encoding e-graphs in Datalog, both encodings are practically orders-of-magnititude slower than `egg`. Constraint-Handlign Rules (CHR) (Frühwirth, 2009) is a potential solution to this problem, as its rules allows more flexible removal of tuples. Moreover, the literature has investigated ways to encode the optimal implementation of union-find in CHR (Schrijvers and Frühwirth, 2006), which is perhaps the most critical step in encoding an e-graph. However, I did not pursue this approach for a long time, since as far as I am aware, available implementations of CHR either misses important features, or is obsure and difficult to use.

Through out this chapter, we will use a very classical equality saturation program, namely associativity and commutativity (AC rules) of the + operator, as our example. The (pseudo)code in Listing 2.1 shows how this can be defined in a library like `egg`. As a baseline, it takes less than one second for `egg` to conclude that $\sum_{i=1}^{8} v_i$ is in the same e-class as $\sum_{i=8}^{1} v_i$. For our Datalog encoding, we did not expect it to be as efficient as highly specialized e-graph frameworks like `egg`. In fact, even the best encodings presented in this chapter are not capable of proving the above equivalence, although it is not unimaginable that a customized Datalog engine can be specialized for our e-graph encodings and therefore more efficient. Moreover, for each of our encodings, it is either the case that there are more or less overheads that will not been seen in a sensible e-graph impelementation, or we have to do some delicate hacking into the Datalog engine that the engine impelementers may be surprised about. Therefore, in some sense, our attempts to encode e-graphs in Datalog is

unsatisfactory. However, as we will see, there are many shining gems we find during this journey.

Listing 2.1: The example equality saturation program used in this chapter.

```
// Enum declaration
define_language! {
    enum Expr {
        Add(Id, Id),
        Var(i64),
    }
}
// Rewrites
let rewrites = vec![
    rw!("(+␣?x␣?y)" => "(+␣?y␣?x)");
    rw!("(+␣(+␣?x␣?y)␣?z)␣=>␣"(+ ?x (+ ?y ?z))");
];
```

2.1 *Encoding E-graphs in Soufflé*

2.1.1 Background

Soufflé is a modern, efficient Datalog engine that has wide applications in program analyses (Antoniadis et al., 2017; Jordan et al., 2016; Hu et al., 2021). While sticking to the dogma of monotonicity, Soufflé has been extended with a diverse range of extensions to both make it easier to program program analyses tasks and faster to run these tasks. These extensions are amenable to the core theory of Datalog, e.g., monotonicity andsemi-naïve evalution. (suppose the user does not break the assumptions)[1]. We sketch some of these extensions

[1]With the exception of termination guarantees of pure Datalog. Similar to programs in many other practical Datalog engines, Soufflé programs may not terminate since they are allowed to populate new values, which is useful in practice.

that are used in our encoding below:

Algebraic Data Types

Soufflé supports algebraic data types (ADTs) as columns. For example, the program below below declares an Abstract Syntax Tree of the example in Listing 2.1 in Soufflé and populates the term $v_1 + v_2$ in relation R:

```
.type Id = Add {x : Id, y : Id}
    | Var {n : number}
.decl R(Id).
R($Add($Var(1), $Var(2))).
```

Internally, Soufflé keeps a record table for ADTs, where each tuple has a unique reference id, the branch id for its constructor, and the field values. Therefore, the encoding is very similar to the one used in relational e-matching, with the difference being in relational e-matching, different branches of an AST is represented as different tables, not different ids within the same table. This encoding allows Soufflé to perform efficient join over ADTs. The reader may wonder why we still use ADTs while we can simulate the same features with the relational encoding *a la* the relational e-matching paper. In fact, we use both: ADTs are specifically used in a skolemizing fashion, i.e., we use ADTs as a handy way to creating new e-class ids. For example, `$Add(x, y)` represents the "natural" e-class id of the e-node with symbol `Add` and children x and y. Other approaches to creating new e-class ids include using the hash of the e-nodes, which we used for Rel.

Equivalence relations

Equivalence relations are widely used for different program analyses tasks, such as Bitcoin user group analysis (Reid and Harrigan, 2013) and points-to analyses (Kastrinis et al., 2018; Steensgaard, 1996). While directly writing these equivalence relations as transitive, reflexive, symmetric rules are highly inefficient, data structures like union find (Tarjan, 1975) can make

reasoning about equivalence orders of magnititude faster. Soufflé provides a built-in support for equivalence relations named `eqrel`. A relation declared as `eqrel` will always satisfy the equivalence rules and is implemented internally using union-find. `eqrel` is designed to be highly parallelizing, and it compactly representes the equivalence relation in linear space, while it takes up to quadratic space to represent it directly.

Subsumptions

Subsumption (Köstler et al., 1995) is the idea that when one tuple is subsumed by another tuple semantically, it does not hurt to remove the subsumed tuples. For example, when computing the shortest paths between pairs of vertices in a graph, one may only care about the shortest paths. Consider the following Datalog program that computes the shortest path:

```
p(x, y, c) :- e(x, y, c).
p(x, y, c) :- p(x, z, cp), e(z, y, ce), c = cp + ce.
sp(x, y, c) :- v(x), v(y), c = min c : p(x, y, c).
```

This program will compute all possible paths between pairs of vertices, before aggregating over the paths to derive the shortest paths. This is inefficient compared to the standard shortest path algorithms like Dijkstra's algorithm. Worst, when the graph contains (even positive) cycles, these rules may not terminate, because there are infinitely many paths. Subsumption addresses this issue by allowing the deletion of paths that are knwon to be not optimal, i.e., those non-shortest paths:

```
sp(x, y, c) :- e(x, y, c).
sp(x, y, c) :- sp(x, z, cp), e(z, y, ce), c = cp + ce.
sp(x, y, c1) <= sp(x, y, c2) :- c1 >= c2.
```

The last rule defines a partial order on `sp` and says that tuple `sp(x, y, c1)` will be subsumed by tuple `sp(x, y, c2)` if `c1` is less than or equal to `c2` (note subsumption is a reflexive relation). Operationally, the "reduced set" will be computed after each iteration of evaluation

according to the subsumptive rules. Köstler et al. (1995) developed a rigourous theory of subsumptions in Datalog and proved its soundness. Finally, other approaches are proposed based on semirings (Abo Khamis et al., 2022b,a) and lattices (Madsen et al., 2016). For example, the Rel language, introduced in Section 2.2, is based on the semiring approach.

In our encoding, we use subsumptions to remove obsolete information. For example, e-classes are being constantly merged, updated, and canonicalized, which will cause e-nodes to be canonicalized from time to time. This leads to the existence of multiple representations of the same e-node, with only one being the canonical at any time. Keeping these stale e-nodes will explode the e-graph. Instead, we can define a partial order over the e-graphs so that all stale e-nodes are subsumed by their canonical version and let subsumptions to clean up the stale e-nodes. We will discuss this in details in Section 2.1.4.

User-defined functors

While Soufflé provides a rich set of primitive operators, it further provides the flexibility by allowing the users to bring their own functions, which Soufflé calls user-defined functors. To declare a user-defined functor, the user defines its implementation in a C++ program and link it during the execution of the Soufflé program. Some of the encodings use the user-defined functors to make `eqrel` more flexible (Zucker, 2022b,a). Compared to the standard usage, we use user-defined functors in a rather wild way, following Zucker (2022b) (see Section 2.1.5).

Aggregations

Finally, Soufflé supports stratified aggregations, which is a standard extension to Datalog. In other words, Soufflé accepts programs where aggregation operators like `max`, `min`, and `sum` does not transitively refer to themselves (i.e., are not recursive). The stratification requirement is crucial to the soundness of the extension because it guarantees that the rules are monotonic. Below is an example that does not satisfy the stratification:

```
R(x) :- x = 1.
```

```
R(c + 1) :- c = max x : R(x)
```

Aftr the first iteration, the database D_1 will contain only $R(1)$. In the second iteration, because the second rule fires, the database D_2 will be $\{R(1), R(2)\}$. However, in the third iteration, the application of the second rule to D_2 will yield $R(3)$, and $R(2)$ that used to exist in D_2 is now found nowhere, which breaks monotonicity.

That being said, there are Datalog programs that break monotonicity, yet are still (semantically) monotonic (e.g., one with subsumptions). We use recursive aggregations throughout in our encoding, because it is semantically clear that e-graphs are growing in a monotonic way. Soufflé does not support recursive aggregations by default, so we pass the `--disable-transformers=SemanticChecker` flag to Soufflé to disable the semantic check. By doing this, we entered the dangerous land of Soufflé since all the assumptions checked by the semantic checker could be violated. This could also have performance implications: each single aggregation is fully computed using linear scan every time instead of incrementally maintained, since the design of Soufflé does not expect recursive use of aggregations. When aggregations are stratified, this is fine because all the aggregations are "one-shot", while when aggregations are used recursively, this means that the aggregations will require repeated linear scans of aggregated relations. This can be prohibitively expensive, and we mitigate this issue with more hacking into Souffé's internals.

2.1.2 A Naïve Encoding

Our first encoding is inspired by the denotation of e-graphs: an e-graph represents a set of terms and a congruence relation over them. We can use the relational representation in the relational e-matching paper to represent the set of terms, and use `eqrel` to represent the congruence relation over terms.

```
.type Id <: number
.decl add(I : Id, I : Id, id : Id)
.decl var(x : number, c : Id)
```

```
.decl eql(x : Id, y : Id) eqrel
add(y, x, yx), eql(xy, yx)
  :- add(x, y, xy), yx = ???.
add(x, yz, x_yz), add(y, z, yz)
  :- add(x, y, xy), add(xy, z, xy_z),
     yz = ???, x_yz = ???.
```

Here we are using integers to represent e-class ids, but we run into an issue: we do not know how to refer to the e-class id of (potentially) new e-classes. For example, in the commutativity rule, it is not obvious what value should we assign to `yx`. One approach is to take the hash function of its children e-class ids, yet it may lead to collision (with a relatively small probability). Instead, we took inspiration from the skolemized chase (Benedikt et al., 2017) and use ADTs to represent the "natural" e-class id of the e-node, e.g., the id of $x + y$ is `$Add(x, y)`.

```
.type Id = Add {x : Id, y : Id}
         | Var {n : number}
.decl add(I : Id, I : Id, id : Id)
.decl var(x : number, c : Id)
.decl eql(x : Id, y : Id) eqrel
add(y, x, yx), eql(xy, yx)
  :- add(x, y, xy), yx = $Add(y, x).
add(x, yz, x_yz), add(y, z, yz)
  :- add(x, y, xy), add(xy, z, xy_z),
     yz = $Add(y, z) x_yz = $Add(x, yz).
```

The above program describes the way we perform rewrites in Soufflé, where the idea is general across the encodings presented in this section.

However, this is not a complete e-graph implementation yet. First, it does not represent the whole term space, which will miss potential rule firing. For example, suppose the databse

has `{add(a, b, c1), add(c2, d, e), eql(c1, c2)}`, because `c1` and `c2` are equivalent, the associativity rule should be fired. However, the rewrite rules does not syntactically match. There are two solutions in this naïve encoding. First, we can modify the rewrite rules, e.g., we can rewrite the associativity rule to be:

```
add(x, yz, x_yz), add(y, z, yz)
  :- add(x, y, xy1), eql(xy1, xy2), add(xy2, z, xy_z),
     yz = $Add(y, z) x_yz = $Add(x, yz).
```

Note that the `eql` relation is quadratic in size, so this rewrite rule will be drastically slower than the original one. Alternatively, it is also possible to simply populate the whole term space, with the following rules.

```
add(x1, y, c) :- add(x, y, c), eql(x, x1).
add(x, y1, c) :- add(x, y, c), eql(y, y1).
add(x, y, c1) :- add(x, y, c), eql(c, c1).
num(x, c1) :- num(x, c), eql(c, c1).
```

As an e-graph can represent exponentially many terms, this could be even slower. Finally, after fixing the above missed firing, there is one last missing piece: `eql` is not a congruence relation yet. For it to be congruent, the following rules need to be added (assuming our encoding populates the entire term space):

```
eql(c1, c2) :- add(x, y, c1), add(x, y, c2).
eql(c1, c2) :- num(x, c1), num(x, c2).
```

If the term space is not explicitly represented, we could add one more indirection to our congruence rule via the `eql` relation as we did for rewrites:

```
eql(c1, c2) :- add(x1, y1, c1), add(x2, y2, c2),
               eql(x1, x2), eql(y1, y2).
```

2.1.3 An Encoding with the Leader Relation

The naïve encoding is not really an e-graph. It is more like an explicit way of encoding the congruence relation and rewrites over it. An e-graph is a particular data structure that makes the congruence relation efficient. To make the congruence relation highly efficient, e-graph aggressively canonicalizes its e-nodes: In a valid e-graph, only canonicalized e-nodes and e-class will exist, so there won't be two different representations of semantically equivalent e-nodes and e-classes. This enables efficient matching in e-graph, since we only need to match on the canonical representation. While in our naïve encoding, we need to either populate all possible representations of an e-node, or we match with a layer of indirection. Therefore, a natural idea is to also do canonicalization in the Datalog encoding.

To canonicalize, we first need to pick a canonical id for each e-class. Here, we pick the canonical id to be the min of all ids (converted from the pointer to the ADT) in this e-class, which we call the leader. The relation that maps an arbitrary id to its leader can be straightforwardly defined as follows:

```
.decl leader(a : Id, b : Id)
leader(a, as(b, Id)) :-
    eql(a, a), // constraint a to be an e-class id
    b = min b1 : { eql(a, b), b1 = as(b, number) }.
```

The leader relation has a similar purpose as the `find` operation of the union-find data structure, as both return the canonical representation of an e-class id and can be rather efficiently maintained under updates to the congruence relations. However, the union-find data structure is significantly more efficiently, since it does not instantiate all the `(id, canonical-id)` pairs, which the leader relation does. Therefore, when an update to the congruence relation happens, the leader relation may update all the $O(N)$ pairs in worst case. Note there are two intricacies to this statement. First, in the encoding presented in this section, no tuples are updated in-place. Rather, new tuples are populated and the old tuples will not be removed. Section 2.1.4 will introduce a way to delete old tuples via

subsumption. Second, the updates to the leader relation is not immediately performed when the congruence relation is updated. Rather, the leader relation is re-computed iteration by iteration in a batched fashion. This may amortize the cost of updating the relation. Overall, the declarative nature of Datalog makes it more complicated to reason about the run-time behavior, in particular the time complexity, of the leader relation. That being said, the declarativeness makes reasoning about the correctness easier.

With the leader relation, we populate canonicalized e-nodes for each e-node:

```
add(x1, y1, c1) :- add(x, y, c),
    leader(x, x1), leader(y, y1), leader(c, c1).
num(x, c1) :- num(x, c), leader(c, c1).
```

Compared to the naïve encoding, this does not populate the whole term space, but only a space that include all e-nodes that are once canonical. As a result, it may contain e-nodes that are not necessarily canonical (but used to). This means extra spaces are needed for these non-canonical e-nodes, and redundant matching may be performed on them. Moreover, the leader relation is monotonically computed, meaning that the leader relation may record all the historical leaders of an e-class id. Yet this is still far less work than our naïve encoding, where we represent every term explicitly. In terms of the actual performance, this encoding is capable of saturate the e-graph with initial expression $\sum_{i=1}^{N} v_i$ under AC rules for $N = 5$ in 0.37 second and $N = 6$ in two minutes, while the naïve encoding does not terminate in three minutes.

As a final note, compared to the naïve encoding, this encoding is no longer stratified: `leader` aggregaets over `eql` and populates `add`; `add` is used in rewrite rules, which updates `eql`. Therefore, `leader` can update the relation it aggregates over, which is not sound in general. Starting at this section, all the encodings rely on the monotonicity of e-graphs to justify the soundness.

2.1.4 Optimizing the leader relation with subsumptions

One of the inefficiencies of the encoding presented in Section 2.1.3 is from the redundant e-nodes and leader entries that was once up-to-date (but no longer). Ideally, one will want to get rid of these stale e-nodes and leader entries, yet plain Datalog does not removal of tuples for the sake of monotonicity. Fortunately, subsumption allows us to remove subsumed tuples given a partial order. Particularly, since we define the leader of an e-class id to be the smallest id that is equivalent to it, the partial order over e-class ids can be straightforwardly given by $<$ within the same equivalence class. Therefore, the following rule can be added to remove stale leader entries when a "better" leader is found:

```
leader(x, y1) <= leader(x, y2) :-
    as(y1, number) <= as(y2, number).
```

Similarly, to canonicalize e-nodes:

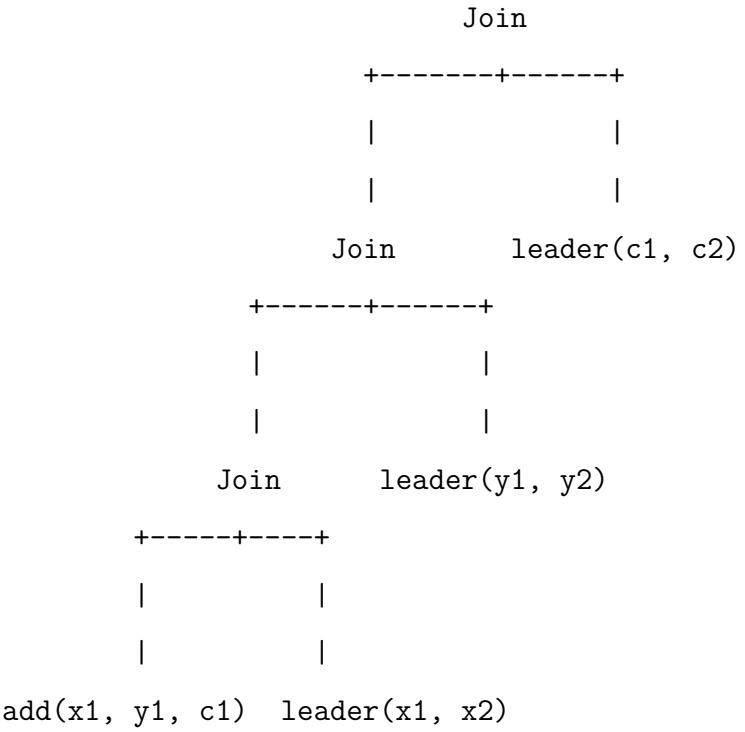
```
add(x1, y1, c1) <= add(x2, y2, c2) :-
    leader(x1, x2), leader(y1, y2), leader(c1, c2).
var(x, c1) <= var(x, c2) :-
    leader(c1, c2).
```

However, the first rule, which canonicalize `add`, is extremely slow in Soufflé due to poor query planning. It will perform a cross product over the `add`×`add`, which is not acceptable. Therefore, we need to instead specify a manual query plan for this rule:

```
add(x1, y1, c1) <= add(x2, y2, c2) :-
    leader(x1, x2), leader(y1, y2), leader(c1, c2).
    .plan 0:(1, 3, 4, 5, 2),
          1:(1, 3, 4, 5, 2),
          2:(1, 3, 4, 5, 2),
          3:(1, 3, 4, 5, 2),
          4:(1, 3, 4, 5, 2),
```

```
5:(1, 3, 4, 5, 2)
```

The following query plan is specified:



Since the leader relation is (almost) an bijection, this join plan can be executed in (almost) linear time. These two rules decreases the runtime from two minutes to 22 seconds for $N = 6$ in the above benchmark, yielding a 6× speedup.

Finally, as a side optimization, so far we have been using `eql(a, a)` enumerate through e-class ids. It turns out Soufflé's `eqrel` will enumerate the (delta of the) `eql` relation for this operation, which is super linear to the size of e-class ids. To get rid of this overhead, we explicitly represent all the ids:[2]

```
.decl ids(a : Id)
ids(x) :- add(_, _, x).
```

[2]Readers may be curious why we still keep the `eql` atom of leader computations. In fact, the author is curious as well—deleting it makes the output unsound. The author speculates this is to make sure that `leader` is dependent on `eql`, which Soufflé cannot infer otherwise. Alas, too many hacks.

```
ids(x) :- var(_, x).
leader(a, as(b, Id)) :-
    ids(a), eql(a, a),
    b = min b1 : { eql(a, b), b1 = as(b, number) }.
```

This makes sure enumerating ids will run linearly in the size of e-class ids and further reduces the run time to 17 seconds.

2.1.5 Efficient computation of leaders using reflections

With the subsumption-based optimiaztion in the last section, we are able to keep only canonicalized e-nodes and e-classes, which makes the rule rewriting even more efficient. However, benchmarking the program from the last section shows that the majority of time is spent in computing the leader relation, even though we have made sure e-class enumeration takes only linear time. Therefore, our last optimization for the Soufflé encoding of e-graph will focus on rules that compute the leader relation.

Let us begin by revisiting the leader rules:

```
leader(a, as(b, Id)) :-
    ids(a), eql(a, a),
    b = min b1 : { eql(a, b), b1 = as(b, number) }.
leader(x, y1) <= leader(x, y2) :-
    as(y1, number) <= as(y2, number).
```

The first rule is extremely slow. Inspecting into the generated IR shows that for every `a`, Soufflé will enumerate through the `eql` relation indexed by `a` to find the smallest id, which is a linear operation. Moreover, even though the smallest id stays the same within the same equivalence class, Soufflé does not know that. As a result, an equivalence class of size n will need to recompute `min` n times. Therefore, rule can take up to quadratic time at each iteration.

To address the second issue, I follow Zucker (2022b), reflecting the internals of `eqrel` of Soufflé out to programs written in Soufflé. Admittedly, this is very hacky and dangerous, but it gives us the last speedup for our Soufflé encoding. To implement this optimization, we first add support for the `FIND` operation, which finds the internal canonical representations of members of the `eql` relation, to our program. A user-defined functor called `FIND` is declared in our Soufflé program, using Soufflé compiler to generate C++ code, and text-replace each call to `FIND` to an call to the internal union-find data structure.

`FIND` allows us to save work by only allowing the canonical id to perform aggregation, and any other id `a` should instead derive its leader by consulting the leader of `FIND(a)`. This ensures that every e-class will perform only one aggregation over its members:

```
leader(a, as(b, Id)) :-
    ids(a), eql(a, a),
    a = FIND(a), b = min b1 : { eql(a, b), b1 = as(b, number) }.
leader(a, b) :-
    ids(a), eql(a, a), a != FIND(a), leader(FIND(a), b).
```

This effectively makes the computation of the leader relation in each iteration linear in the size of ids, since every id will be enumerated only once by the `ids` relation and once by the aggregation.

Finally, if the internals of Soufflé have already provided us with a canonical representation, why would we even bother to compute leader at all?

```
leader(a, FIND(a)) :- ids(a), eql(a, a).
leader(x, y1) <= leader(x, y2) :-
    eql(y1, y2),
    y2 = FIND(y1).
```

This is our ultimate encoding of e-graphs in Soufflé, being able to saturate $\sum_{i=1}^{6} v_i$ under AC rules in under one second, and $\sum_{i=1}^{7} v_i$ within 20 seconds, which all other encodings presented in this section times out.

2.2 Encoding E-graphs in Rel

The Soufflé encoding of e-graphs is concise and efficient. However, it relies on some of the critical features of Soufflé, such as subsumptions and built-in equivalence relations, without which the encoding will be orders of magnitude slower (e.g., the naïve encoding in Section 2.1.2). Moreover, the design of the Soufflé encoding is also limited by some of the restrictions Soufflé imposes. For example, strict monotonicity means that tuples existing in the last iteration will stay here unless subsumption explicitly removed it.

In this section, we consider this same problem of encoding e-graphs in Datalog from a quite different spot in the design space. Rel does not support subsumptions nor built-in equivalence relations, yet it eases general-purpose programming by sacrificing monotonicity. Rel does not check for monotonicity during the compilation, and the evaluation algorithm by default does not perform semi-naïve evaluation nor keep old tuples. Instead, Rel iteratively applies the rules to the lastest database instance to derive the input to the next iteration, so if a tuple t is computed at iteration $n-1$ but not at iteration n, where the fixpoitn is reached, the resulting database instance will not contain t. The oversimplified evaluation algorithm is as follow:

```
def eval_rules(edb, rules):
    db = edb
    do
        db = apply_rules(rules, db)
    while db does not change
```

This is a fundamental difference between Rel and Soufflé: in Soufflé, even if the rules are not monotonic, one has to explicitly use subsumptions to remove an old tuple. The oversimplified evaluation algorithm of Soufflé is as follows:

```
def eval_rules(edb, rules):
    db = edb
```

```
do
    new_db = apply_rules(rules, db)
    db = db + new_db
while delta does not change
```

As we have seen, this can sometimes be a headache because one need to explicitly remove obsolete tuples. In comparison, Rel's non-monotonic semantics allows us to write iterative algorithms that are not monotonic (with regard to tuple inclusion) much easier.

Rel also has an expressive set of language features: relations can be defined in place using comprehension syntax, are easily composible via comma (,) operator (for cartesian product) and semicolon (;) operator (for union), and can be used as arguments to higher-order functions like `hash`; universal and existential quantifiers can be nested; and there are rich syntactic sugars for relational programming, e.g., `R[a]` means `b: R(a, b)`. This section will not attempt to cover all the features of Rel; instead, we will introduce each language feature of Rel as we use them.

2.2.1 Representing terms

The first issue we address is the representation of terms in Rel: for a relation R, we say e-node $f(x_1, \ldots, x_n)$ with id i is in R if $R(: f, x_1, \ldots, x_n, i)$. $: f$ is a compile-time literal that is compiled into the metadata of the relation, such that $R : f$ is a "sub-relation" of R and can be visited as $R[: f]$ as a normal get-by-key operation. We also define the "natural" e-class id of an e-node, which is also the initial e-class id of an e-node to be the hash value of tuple $(: f, x_1, \ldots, x_n)$. The following Rel code populates $\sum_{i=1} nv_i$

```
def expr_prep = hash[{:var, range[1, N, 1]}]
def expr_prep = hash[{:add, expr_prep:var[1], expr_prep:var[2]}]
def expr_prep = hash[{:add,
                      expr_prep:add[_, expr_prep:var[n]],
                      expr_prep:var[n+1]
```

```
                        from n where n<N}]
def expr = expr_prep
```

`hash` is a higher-order function that takes a relation with arity n, and produce a new relation with arity $n+1$, where the first n columns are from the input relation, and the last column is the hash value of each tuple.

2.2.2 Representing the Union-Find Data Structure

One critical distinction between the encoding in Rel and in Soufflé is that Rel does not have built-in efficient equivalences. Naïvely encoding the equivalence relation as a transitive, reflexive, and symmetric relation is possible, but slow. It turns out that it is possible to encode the union-find data structure in Rel, by declaratively stating the properties of each component.

There are three key components for a union-find data structure in our encoding: `link`, which link children and parents in the union-find tree; `find`, which returns the canonical id (i.e., root) for the given id; and finally `merge`, which is a list of pairs that records the history of operations over the union-find. In each iteration, the union-find (namely `find` and `link`) will be built according to `merge` from scratch. Similar to our Soufflé encoding, the root is the id with smallest value in an e-class.

Let us start by defining `find`: the output of `find` will be the minimum of itself and `find` of its parent. This gives the following declarative rule:

```
def find = a in last[expr]: to[min[fr[a]; fr[find[link[a]]]]]
```

where `last[expr]` gives all the e-class ids and `fr` and `to` are helper functions to convert from and to hash values:

```
@inline def to = uint128_hash_value_convert
@inline def fr = hash_value_uint128_convert
```

Similarly, there are three cases for `find`: because of path compression, the `link` of an id is either itself (when its root), the `find` of its parent, or `find[b]` when the id is equal to `find(a)` for some merge history `merge(a, b)`. This gives the following rule.

```
def link = l in last[expr]: to[min[
    fr[l];
    fr[find[l]];
    (fr[find[b]] from a, b where l = find[a] and merge(a, b));
    (fr[find[b]] from a, b where l = find[a] and merge(b, a))]]
```

In fact, `link` and `find` will converge at the fixpoint: By definition of `find`, (1) `find[a] <= find[link[a]` and (2) `find[a] <= a`, and by definition of `link`, (3) `link[a] <= find[a]` and (4) `link[a] <= a`. By (3) and (1), we have

```
link[a] <= find[a] <= find[link[a]]
```

By (2), `find[link[a]] <= link[a]`. Thus, `link[a] <= find[a] <= link[a]`, which implies `find[a] = link[a]` for all `a`.

2.2.3 Representing Rewrites

Finally, we turn our attention to the actual representation of rewrites. This is easier to do than in the Soufflé encoding, because we do not need to worry about removal of obsolete tuples (e.g., using subsumptions).

We use three relations to represent a term for different purposes. All three relations will have the same first n arguments but differ in their last column, which is used for representing different ids.

- The last column of `expr` is the natural id of the e-node, i.e., the hash value of its function symbol and its children;

- The last column of `rels` is the canonical id of that e-node, given by the `find` procedure. Moreover, `rels` will only contain canonical e-nodes. This is the relation we do e-matching on;

- Finally, the last column of `rels_todo` documents e-class ids that need to be merged later.

We first define `rels`, which is the easiest among the three representations. Essentially, `rels` canonicalize each e-node from `expr`:

```
def rels:add = find[x], find[y], find[xy]
               from x, y, xy
               where expr:add(x, y, xy)
def rels:var = v, find[c] from v, c
               where expr:var(v, c)
```

The definition of `expr` and `rels_todo` are query dependent: besides the initially populated tuples in `expr`, all the tuples in `expr` and `rels_todo` are directly produced by rewrites. First, for each rewrite, we need to insert the (potentially) new e-nodes (with its natural ids) into the `expr` relation:

```
// comm
def expr = hash[{:add, (x, y: rels:add(y, x, _))}]

// assoc
def expr = hash[{:add, (y, z: rels:add(_, y, xy), rels:add(xy, z, _)
                               from xy)}]
def expr = hash[{:add, (x, yz: rels:add(y, z, yz), rels:add(x, y, xy),
                                rels:add(xy, z, _)
                                from xy, y, z)}]
```

For commutativity, we populate $x + y$ for each $y + x$ found (in the canonical database). Similarly for associativity, we populate $y + z$ and $x + (y + z)$ for each $(x + y) + z$ found.

For `rels_todo`, we essentially insert into it the e-class ids from the left-hand side of the rewrite rules, which will be the new e-class ids for the right-hand side pattern, processed by the union-find.

```
def rels_todo:add(x, y, yx) = rels:add(y, x, yx)
def rels_todo:add(x, yz, xy_z) =
    rels:add(y, z, yz),
    rels:add(x, y, xy),
    rels:add(xy, z, xy_z)
    from xy, y, z
```

We tie the knots and define the `merge` function based on `rels_todo`. In particular we merge ids from `rels_todo` with the natural id (which is known to be in the id).

```
def merge = id1, id2: expr:add(x, y, id1), rels_todo:add(x, y, id2) from x, y
```

But we are not done yet: equivalences may be introduced by canonicalization. Suppose `add(a, b, c)` and `add(d, e, f)` are present in the database and later `d` is canonicalized to `a` and `e` to `b`. This will effectively make the canonical database `rels` to have two "canonical" ids for the same e-node, which we need to merge:

```
def merge = id1, id2: rels:add(x, y, id1),
                      rels:add(x, y, id2),
                      id1 != id2
                      from x, y
```

2.3 Miscellaneous

This chapter summarizes the work done during the first half of my master. Among my collaborators, I would particularly like to thank Philip Zucker for his inspirational blog

post on this topic and for insightful discussions. I want to also thank Martin Bravenboer for introducing me to the Rel language and later for his guidance during my internship at RelationalAI. It is truly an honor to work with my colleagues at RelationalAI.

Chapter 3

EGGLOG

This chapter presents an semi-formal introduction on egglog, a prototype language that attempts to generalize both Datalog and egg. Currently, only a partial implementation of the described egglog language exists, with some deviations from the language presented here.

Why egglog? The motivation behind egglog is to find a good model for relational e-graphs that can take full advantage of (1) performance of relational e-matching and (2) expressiveness of Datalog, while (3) being compatiable with egg as well as (4) efficient. This is the first approach described at the beginning of Chapter 1. Compared to the second approach, which as we saw is limited in the kind of optimization it can perform, this approach is more principled and fundamental.

Egglog is a dialect of Datalog, so it supports various reasoning expressible in Datalog. A rule has the form `head1, ..., headn :- body1, ..., bodyn`. For example, below is a valid egglog program:

```
rel link(string, string) from "./link.csv".
rel tc(string, string).

tc(a, b) :- link(a, b).
tc(a, b) :- link(a, c), tc(c, b).
```

However, Datalog by itself is not that interesting. So for the first part of this chapter, we will instead focus on the extensions that make egglog interesting. Next, we will give some examples and show why egglog generalizes egg. We will also try to develop the operational and model semantics of egglog.

3.1 Introduction to Egglog

3.1.1 User-defined sorts and lattices

In egglog, every value is either a (semi)lattice value or a sort value. Lattices in egglog are algebraic structures with a binary join operator ($\vee$) that is associative, commutative, and idempotent and a default top $\top$ where $\top \vee e = \top$ for all e. For example, standard types like `string`, `i64`, and `u64` in egglog are in fact trivial lattices with $s_1 \vee s_2 = \top$ for all $s_1 \neq s_2$. In egglog, $\top$ means unresolvable errors. Users can define their own lattices by providing a definition for lattice join.

Similarly, users can define sorts. Unlike lattices, sorts are uninterpreted. As a result, sort values can only be created implicitly via functional dependency. We will go back to this point later.

3.1.2 Relations and Functional Dependencies

Relations can be declared using the `rel` keyword. Moreover, it is possible to specify a functional dependency between columns in egglog. For example,

```
sort expr.
rel num(i64) -> expr.
```

declares a sort called `expr` and a `num` relation with two columns `(i64, expr)`. In the `num` relation, each `i64` uniquely determines the remaining column (i.e., `num(x, e1)` and `num(x, e2)` implies `e1 = e2`). The `num` relation can be read as a function from `i64` to values in `expr`. Similar declarations are ubiquitous in egglog to represent sort constructors.

As another example,

```
rel add(expr, expr) -> expr.
```

declares a relation with three columns, and the first two columns together uniquely determines the third column. This represents a constructor with two `expr` arguments.

Users can introduce new sort values with functional dependencies. Example:

```
num(1, c). % equivalently, num(1, _).
num(2, d).
add(c, d, e) :- num(1, c), num(2, d).
```

This program is interesting and its semantics deviates from the one in standard Datalog. In standard Datalog, this program will not compile because variable `c` in the first rule, `d` in the second rule, and `e` in the third rule are not bound. However, this is a valid program in egglog. Thanks to functional dependency, variables in the head do not necessarily have to be bound in the bodies. Variables can be unbound as long as they can be inferred from the functional dependency. The above egglog program is roughly equivalent to the following Datalog program:

```
num(1, c) :- !num(1, _), c = new_expr().
num(2, d) :- !num(2, _), d = new_expr().
add(c, d, e) :- num(1, c), num(2, d), !add(c, d, _), e = new_expr()
```

Negated atoms like `!num(1, _)` is necessary here because otherwise it will inserts more than one atoms matching `num(1, _)`, which violates the functional dependency associated to the relation.

The above example egglog program can also be written into one single rule with multiple heads:

```
add(c, d, e), num(1, c), num(2, d).
% roughly equivalent to
% add(c, d, e), num(1, c), num(2, d) :- !num(1, _), !num(2, _),
%                                       c = new_expr(),
%                                       d = new_expr(),
%                                       !add(c, d, _),
%                                       e = new_expr().
```

Egglog also supports the bracket syntax, so the last program can be further simplified to:

```
add[num[1], num[2]].
```

The bracket syntax will implicitly fill the omitted column(s) with newly generated variable(s). If the atom is nested within another term, the nested atom will be lifted to the top-level, and the generated variable(s) will take the original position of the atom. Here is another example of the bracket syntax:

```
ans(x) :- xor[xor[x]].
% expands to
% ans(x) :- xor[y, z], xor(x, y, z)
% which expands to
% ans(x) :- xor(y, z, _), xor(x, y, z)
% this rule can be thought as
%   for any expr x, y where \texttt{y xor (x xor y)}
%   is present in the database, collect x as the result.
```

Finally, in equational reasoning *a la* egg, it is common to write rules like "for every `(a + b) + c`, populate `a + (b + c)` on the right and make them equivalent". This rule will look like the following:

```
add(a, add[b, c], id) :- add(add[a, b], c, id).
```

Egglog further has a syntactic sugar for these equational rules: `head := body if body1 ... bodyn` where both `head` and `body` should use the bracket syntax and omit the same number of columns. The if clause can be omitted. Egglog will expand this syntactic sugar by unfolding the top-level bracket in `head` and `body` with the same variable(s):

```
add[b, a] := add[a, b].
% unfolds to add(b, a, id) :- add(a, b, id).
add[a, add[b, c]] := add[add[a, b], c].
% unfolds to add(a, add[b, c], id) :- add(add[a, b], c, id).
```

```
num[1] := div[a, a] if num(x, a), x != 0.
% unfolds to num(1, id) :- div(a, a, id), num(x, a), x != 0.
```

Note the equational rules may introduce functional dependency violation; for instance, last rule may cause multiple tuples to match `num(1, _)`, yet the first column should uniquely determines the tuple. We will discuss more about how we resolve this kind of violations The essential idea is that, if two sort values are present with the same primary key, then the two sort values must be equivalent, whereas if two lattice values are present with the same primary key, the new, unique lattice value should generalize the two values, i.e., it will be the least-upper bound of those lattice values.

3.1.3 Relations with lattices

The example relations we see so far mostly center around sort values. However, it is also possible and indeed very useful to define relations with lattices:

```
rel hi(expr) -> lmax(-2147483648).
rel lo(expr) -> lmin(2147482647).
```

To define a lattice column, a default value need to be provided in the relation definition. The default value is not a lattice bottom: the bottom means do not exist. Meanwhile, the lattice top means there are conflicts. It is also possible for default value to refer to the determinant columns:

```
rel add1(i: i64) -> i64(i + 1).
```

The column initialization syntax should be reminiscent of C++'s member initializer lists.

In the above example, `lo` and `hi` together define a range analysis for the `expr` sort. This in facts generalizes the e-class analyses in egg. Here are some rules for `hi` and `lo`:

```
hi(x, n.into()) :- num(n, x).
lo(x, n.into()) :- num(n, x).
```

```
lo(nx, n.negated()) :- hi(x, n), neg(x, nx).
hi(nx, n.negated()) :- lo(x, n), neg(x, nx).
lo(absx, 0) :- abs(x, absx).
lo[absx] := lo[x] if abs(x, absx), lo[x] >= 0.
hi[absx] := hi[x] if abs(x, absx), lo[x] >= 0.
lo(xy, lox + loy) :- lo(x, lox), hi(y, loy), add(x, y, xy).
% can be further simplified to
%   lo[xy] := lo[x] + lo[y] if add(x, y, xy)
```

Note here instead of `lo(neg[x], n.negated()) :- hi(x, n).`, we put the `neg` atom to the right-hand side and write `lo(nx, n.negated()) :- hi(x, n), neg(x, nx).` There are some nuanced differences between the two rules. This rule, besides doing what the second rule does, always populates a `neg` tuple for each `hi` tuple even when it does not exist, so the first rule can be viewed as an "annotation-only" version of the first rule, which is usually what we want.

The last example shows e-class analyses in egglog is composable (i.e., each analysis can freely refer to each other). This is one of the reason why we believe egglog generalizes e-class analyses. Moreover, they can also interact with other non-lattice relations in a meaningful way:[1]

```
rel geq(expr, expr).

% ... some arithmetic rules ...

% need to convert to int because they are from different lattices
geq(a, b) :- lo[a].to_int() < hi[b].to_int().
% ... other user-defined knowledge about geq
```

[1]The last rule in this example has a single variable on the left-hand side, but the above mentioned syntactic expansion for `:=` does not apply to this case. The rule is indeed equivalent to `abs(x, x) :- abs[x] if geq(x, num[0])`.

```
% x and abs[x] are equivalent when x > 0
x := abs[x] if geq(x, num[0])
```

Diverging a little bit, it is even possible to write the above rules without using lattice relations:

```
sort bool.
rel true() -> bool.
rel false() -> bool.
rel geq(expr, expr) -> bool.

% for each abs[x] exists, populate geq[x, 0],
% in the hope that later
% it will be "in the same e-class" as true[].
geq[x, 0] :- abs[x]

geq[numx, 0] := true[] if num(x, numx), x > 0.
geq[xx, 0] := true[] if mul(x, x, xx).
% if x > 0 and y > 0 are both equivalent to true,
% then x + y > 0 is also equivalent to true.
geq[xy, 0] := true[]
          if add(x, y, xy),
              geq(x, 0, true[]),
              geq(y, 0, true[])
geq[xxyy, 0] := true[] if add(mul[x, x], mul[y, y], xxyy).

% ... other reasoning rules...
```

```
% if x>0 is equivalent to true,
% every abs[x] in the database should be equivalent to x
x := abs[x] if geq(x, 0, true[])
```

The above program can be seen as implementing a small theorem prover in egglog. Whenever it sees abs[x], a query about `x >= 0` will be issued to the database. If later `x >= 0` is proven to be equivalent to true, a distinguished sort value, `abs[x]` will be put in the same e-class as `x`.

All these rewrite will be very hard to express in egg.

3.1.4 Functional Dependency Repair

FDs can be violated: what if the user introduced two values for the same set of determinant columns? In this case, we need to repair the FDs. We have seen such examples many times in previous sections. For example, rules like `R[x1, ..., xk] := ...` will add new values to `R` indexed by `x1, ..., xk`, and it is likely that there are already other tuples with the same prefix `x1, ..., xk`. These rules may potentially cause violation of functional dependencies. In general, there are two kinds of violations:

- Case 1: If the dependent column is a sort value, egglog will unify the two sort values later in the iteration. We can think of a term of a sort in egglog as a constant in some theories, which refers to some element in the model. But we don't know which element it refers to. However, by repairing functional dependencies, we can get some clues about what the structure will look like. Consider the following program

  ```
  rel add(expr, expr) -> expr.
  rel num(i64) -> expr.

  % add the fact 2 + 1, where the last column is auto-generated.
  add[num[2], num[1]].
  ```

```
% add the fact 2+1, but the last column is add[num[1], num[2]]
% (add[num[1], num[2]] is created on the fly because
% it occurs at the left hand side.)
add(num[2], num[1],
add[num[1], num[2]]).
```

Because now (without repairing) `add[num[1], num[2]]` will contain two rows. The functional dependency is violated. If we think of rewriting under functional dependency as a process of finding a model for the sort, then what do we learn from this violation? We learned that, to respect the functional dependency, the two sort values must be the same thing! Therefore the expr originally referred by `add[num[2], num[1]]` and by `add[num[1], num[2]]` will be treated as the same expr and no longer be distinguishable in egglog! As we will show later, when a egglog program reaches the fixpoint, it produces a valid, minimal model for the relations and the sorts such that the rewrite rules and the functional dependencies are both respected.

- Case 2: What if the dependent column is a regular type as a Rust struct or an integer? Well, we also need to unify them, but in a different way. The idea here is to describe these values with a algebraic structure, which in this case is a lattice. A lattice has a bottom (means does not exist) and a top (means conflicts). Similar to Flix, lattice values will grow by taking the least upper bound of all the violating tuples. In that sense, egglog also generalizes Flix (as is described in the Madsen et al. (2016)).

3.1.5 Seamless Interop with Rust

This proposed extension takes inspiration from recent work on Ascent(Sahebolamri et al., 2022), an expressive Datalog engine that has seamless integration with the Rust ecosystem. One interesting feature of Ascent is that it allows first-class introspection of the column

values. Ascent use this feature to support features like first-class environment (this and the next example are both from page 4 of the Ascent paper; comments are mine):

```
sigma(v, rho2, a, tick(e, t, k)) <--
  sigma(?e@Ref(x), rho, a, t), // the environment rho is enumerated here
  store(rho[x], ?Value(v, rho2)), // rho[x] is used as an index for store
  store(a, ?Kont(k));
```

One thing though is that Ascent allows enumerating structs as a relation with the `for` keyword. For example:

```
sigma(v, rho2, store, a, tick(v, t,k)) <--
  sigma(?Ref(x), rho, store, a, t),
  // enumerating store[&rho[x]]
  for xv in store[&rho[x]].iter(), if let Value(v,rho2) = xv,
  // enumerating store[a]
  for av in store[a].iter(), if let Kont(k) = av;
```

This makes Ascent have a more macro-y vibe, which makes sense since the whole Ascent frontend is based on Rust's procedural macros. However, I think the similar can be easily achieved inside the relational land, so in a full-fledged relational language like egglog, the `for` syntax may not be necessary.

Seamless interop with Rust is in general very powerful. In fact, we have already used this feature a lot. For example, lattices in egglog are structs defined in Rust that implements certain traits. So rules like `hi(x, n.into()) :- num(n, x).`, will call methods in the corresponding struct (e.g., `n.into()`).

In general, these user-defined functions introduced functional dependencies from domains of functions to their range. For example, rule `hi(x, n.into()) :- num(n, x).` can be viewed as `hi(x, n_into) :- num(n, x), into_rel(n, n_into)` with functional dependency from `n` to `n_into`. Advanced join algorithms like worst-case optimal joins can leverage these functional dependencies to optimize the query.

3.2 The Model Semantics of Egglog and its Evaluation

In this section, I will focus on the problem of how to formalize egglog and how to evaluate egglog programs. This section will first give the model semantics of egglog. Then, I will describe rebuilding, an essential procedure for evaluating and maintaining e-graphs, namely rebuilding, in the egglog setting. Finally, we will discuss how egglog's matching procedure can benefit from semi-naive evaluation, a classic evaluation algorithm in Datalog

3.2.1 The Model Semantics

Given a schema S with functional dependencies between columns, we define the set of function symbols $F = \{f_{R,1}, \ldots, f_{R,m} \mid R(c_1, \ldots, c_n) \to (d_1, \ldots, d_m) \in S\}$. The universe U_S is given as the set of terms constructible from constants and function symbols.

From the universe U_S, we define a structure $M = (\cong, \Delta)$, where $\cong$ is a congruence relation over U_S. Given a total order $<$, each congruence relation derives a unique mapping $\lambda_{\cong} : U_S \mapsto U_S$ that maps every term to the smallest term in its equivalence class. Δ is the set of tuples $\{\varphi_R \in \mathcal{P}(U_S^{a_R}) \mid R \in S\}$, where a_R is the arity of relation R. We say $R(t_1, \ldots, t_{a_R}) \in \Delta$ if $(t_1, \ldots, t_{a_R}) \in \varphi_R$. $R(t_1, \ldots, t_{a_R})$ is in M, or $M \models R(t_1, \ldots t_{a_R})$, if $R(\lambda(t_1), \ldots, \lambda(t_{a_R})) \in \Delta_M$. Moreover, we define $t_1 = t_2$ holds in M, or $M \models t_1 = t_2$, if $\lambda(t_1) = \lambda(t_2)$.

We say $M_1 \sqsubseteq M_2$ if $M_1 \models \phi$ implies $M_2 \models \phi$ for any predicate ϕ. $\sqsubseteq$ is a partial order. We further define intersection $(\cong_1, \Delta_1) \sqcap (\cong_2, \Delta_2))$ to be

$$(\cong_1 \cap \cong_2, [\lambda_{\cong_1} \mapsto \lambda_{\cong_1 \cap \cong_2}]\Delta_1 \cap [\lambda_{\cong_2} \mapsto \lambda_{\cong_1 \cap \cong_2}]\Delta_2),$$

where

$$[\lambda_1 \mapsto \lambda_2]\Delta = \{(\lambda_2(u_1), \ldots, \lambda(u_{a_R})) \mid R(t_1, \ldots, t_{a_R}) \in \Delta, \lambda_1(u_1) = t_1, \ldots, \lambda_1(u_{a_R}) = t_{a_R}\}.$$

$\sqcap$ is monotone: we have $M_1 \sqcap M_2 \sqsubseteq M_1$ and $M_1 \sqcap M_2 \sqsubseteq M_2$.

Example 1. *Suppose* $<$ *is the total order of natural numbers,* $M_1 = (\{\{1,2,3\},\{4\}\},\{R(1,4)\})$, $M_2 = (\{\{1\},\{2,3,4\}\},\{R(2,2)\})$, *we have* $\cong_{M_1 \sqcap M_2} = \{\{1\},\{2,3\},\{4\}\}$ *and* $\Delta_{M_1 \sqcap M_2} = \{R(1,4),R(2,4)\} \cap \{R(2,2),R(4,2),R(2,4),R(4,4)\} = \{R(2,4)\}$.

A rule in egglog has the form

$$\exists Y, S_1(Y_1), \ldots, S_m(Y_m) \leftarrow R_1(X_1), \ldots, R_n(X_n).$$

Thanks to the inferrable constraint, we can eliminate the existentially quantifier and the above rule can be rewritten as first-order sentence with only universal quantifiers (TODO: mention why we do this: because we need Y'_1 to uniquely determine $\vec{f_{S_1}}(Y'_1)$ and so forth)

$$\begin{gathered}
\forall X, R_1(X_1), \ldots, R_n(X_n) \rightarrow \\
S_1(Y'_1, \vec{f_{S_1}}(Y'_1)), \ldots, S_m(Y'_m, \vec{f_{S_m}}(Y'_m)), \\
f_{j_1}(Y'_{k_1}) = f_{j'_1}(Y'_{k'_1}), \ldots, f_{j_p}(Y'_{k_p}) = f_{j'_p}(Y'_{p'_1}) \\
f_{x_1}(Y_{y_1}) = X_{z_1,w_1}, \ldots, f_{x_q}(Y_{y_q}) = X_{z_q,w_q},
\end{gathered}$$

Example 2. *Rule* $\exists y, add(\beta,\gamma,y), add(\alpha,y,r) \leftarrow add(\alpha,\beta,x), add(x,\gamma,r)$ *will be rewritten to the first-order sentence*

$$\begin{aligned}
\forall \alpha,\beta,\gamma,x,r. add(\alpha,\beta,x), add(x,\gamma,r) \rightarrow & add(\beta,\gamma,f_{add,1}(\beta,\gamma)), \\
& add(\alpha,f_{add,1}(\beta,\gamma),f_{add,1}(\alpha,f_{add,1}(\beta,\gamma))), \\
& f_{add,1}(\alpha,y) = r.
\end{aligned}$$

An egglog program is a set of rules Γ satisfying the inferrable constraint. M is a model of Γ, denoted as $M \models \Gamma$, if every rule in Γ hold. Every model of a program Γ also respects the functional dependency. Given a program, there always exist a satisfying model, which can be trivially given by $(\cong_\top, \Delta_\top)$, where every term in U_S is equivalent and $\Delta_\top = \{R(c_0,\ldots,c_0) \mid R(\ldots) \rightarrow (\ldots) \in S, c_0 = \min_<(U_S)\}$.

Moreover, there is a smallest model M for every program Γ, given as the intersection of all models $M_{\min} = \bigsqcap_M M$.

Proof. Suppose otherwise that $M_{\min}$ is not a model. Then there must be some grounded rule where the body is satisfied but some atom ϕ in head is not.

- If ϕ has the form $a = b$, for every valid model $M = (\cong, \Delta)$, (a, b) must be in $\cong$, so $(a, b) \in \cong_{\min}$. This is a contradiction.

- If ϕ has the form $R(t_1, \ldots, t_{a_R})$, for every valid model M, $R(\lambda(t_1), \ldots, \lambda(t_{a_R})) \in \Delta_M$. Therefore,

$$\begin{aligned}\Delta_{\min} &\supseteq \bigsqcap_M \{R(\lambda_{\cong_{\min}}(u_1), \ldots, \lambda_{\cong_{\min}}(u_1)) \mid \lambda_{\cong_M}(t_1) = \lambda_{\cong_M}(u_1), \ldots, \lambda_{\cong_M}(t_{a_R}) = \lambda_{\cong_M}(u_{a_R})\} \\ &\supseteq \bigsqcap_M \{R(\lambda_{\cong_{\min}}(t_1), \ldots, \lambda_{\cong_{\min}}(t_1))\} \\ &= \{R(\lambda_{\cong_{\min}}(t_1), \ldots, \lambda_{\cong_{\min}}(t_1))\}\end{aligned}$$

  This implies $M \models R(t_1, \ldots, t_{a_R})$, which is a contradiction.

Therefore, $M_{\min}$ is a model. □

In fact, the result of evaluating an egglog program using the abstract algorithm below, if terminates, is the smallest model of Γ.

```
def run(Gamma):
  M = (Id, {})
  while not fixpoint:
    newM = M
    for r in Gamma:
      for grounded s in match(r, M):
        newM = apply(newM, s.rhs)
    M = newM
```

3.2.2 The Detailed Evaluation Algorithm

Following the idea of the above abstract algorithm, here we present a detailed evaluation algorithm. This algorithm is not yet fully implemented, and it is a sketch of what I think should work. The readers should read this section with a grain of salt, since there is absolutely no correctness guarantees. The evaluation algorithm of egglog programs consists of two parts. The core of the evaluation is the invariant-maintaining rebuilding algorithm, which is inspired both by the rebuilding algorithm of egg and by the evaluation algorithm of the chase. The second part involves matching and applying egglog rules. Applying egglog rules is efficient. In the chase's terminology, thanks to the above mentioned inferrable constraint, rule application in egglog is able to utilize functional dependencies to avoid to generate unnecessary nulls. Moreover, because egglog programs are monotonic computations over the relational database in nature, they can benefit from the semi-naive evaluation algorithm of Datalog. We call this semi-naive matching, which can be seen as a further improvement over relational e-matching (Zhang et al., 2022).

The main algorithm will iteratively apply `batch_rewrite`, which will apply the rules in batches and will return false if no updates to the database is found. After rule firing, `rebuild` will be called to restore the functional dependencies.

```
def run(pats, DB, max_iter):
  for iter in range(max_iter):
    if !batch_rewrite(pats, DB):
      return
    rebuild(DB)
```

Rebuilding

The rebuilding algorithm, shown below, is similar to the one used in `egg`, but generalizes it since it handles functional dependencies beyond congruence:

```
  todo = mk_union_find()
```

```
domain = mk_set()

def union_sort(s1, s2):
  todo.union(s1, s2)
  domain.add_all([s1, s2])

def refresh_todo():
  todo = mk_union_find()
  domain = mk_set()

def on_insert(R, tup):
  # find the tuple by its determinant columns
  orig_tup = R.find_by_determinant(tup.det)
  if orig_tup is None:
    R.insert(tup)
  else:
    # enumerate each dependent column
    for c1 in tup.dep:
      col = c1.col
      c2 = orig_tup[col]
      if col.is_sort():
        s1 = todo.get_or_create(c1)
        s2 = todo.get_or_create(c2)
        union_sort(s1, s2)
      else:
        orig_tup.set_col(col, c1.lat_max(c2))

def normalize(tuple, union_find):
```

```
  return tuple.map(lambda val:
    union_find.get_or_default(val, default = val))

def rebuild(DB):
  while not todo.is_empty():
    # take todo into the local scope
    union_find = todo
    refresh_todo()

    to_remove = mk_set()
    to_insert = mk_set()

    for val in domain:
      for R in DB:
        for col in R.cols:
          for tup in R.index_by(col = col, val = val):
            new_tup = normalize(tup, union_find)
            if new_tup != tup:
              to_remove.add((R, tup))
              to_insert.add((R, new_tup))

    DB.remove_all(to_remove)
    # may trigger on_insert
    DB.insert_all(to_insert)
```

Applying rewrite rules

The rule firing of egglog can be viewed as a combination of rule firing in the chase (for sorts) and in Flix (for lattices)

```
def batch_rewrite(pats, DB):
  to_insert = mk_set()
  for (lhs, rhs) in pats:
    for subst in match(DB, lhs):
      subst = chase(DB, subst, lhs, rhs)
      for (R, atom) in rhs:
        to_insert.add((R, atom.apply(subst)))
  DB.insert_all(to_insert)
  return to_insert.is_empty()

def chase(DB, subst, lhs, rhs):
  shouldContinue = True
  while shouldContinue:
    shouldContinue = False

    for atom in rhs:
      det_vars = atom.get_det_vars()
      if det_vars.is_subset_of(subst.get_domain()):
        shouldContinue = True

        R = DB.get_rel(atom.rel)
        det = det_vars.apply(subst)
        tup = R.find_by_determinant(det)
        for var in atom.get_dep_vars():
          col = var.col
          if var.is_sort():
              if tup is None: continue
              value = tup.get_by_col(col)
```

```
            sort_update(subst, var, value)
          else:
            value = tup is None ? col.lat_init(det)
                                : tup.get_by_col(col)
            lat_update(subst, var, value)

  for var in rhs.get_all_vars():
    if !subst.contains(var):
      assert var.is_sort():
      subst[var] = new_sort_value(var.sort)

def lat_update(subst, var, value):
  if subst.contains(var):
    subst[var] = subst[var].lat_max(value)
  else:
    subst[var] = value

def sort_update(subst, var, value):
  if subst.contains(var):
    union_sort(subst[var], value)
  else:
    subst[var] = value
```

Semi-Naive Matching

One of the bottleneck in evaluating egglog programs is matching the left-hand side. Since we are matching over a relational representation of the e-graphs, we are already doing is already relational e-matching. However, we can go one step further: Let `DB'` be the database of tuples that are not touched in the current iteration of rewrite. `DB'` by itself will not

produce any interesting new tuples; it has to join with newly generated tuples (i.e., the delta database). This is exactly the semi-naive evaluation algorithm of Datalog. We call this similar optimization in egglog semi-naive matching. This optimization will be tricky to do over e-graph's DAG representation, yet is fairly obvious in egglog's full-fledged relational representation.

Chapter 4

RELATED WORK

4.1 E-graphs

E-graphs are first introduced in Greg Nelson's seminal thesis (Nelson, 1980) in late 1970s as a way of effectively deciding the theory of equalities. A more efficient algorithm is later introduced by Downey et al. (1980) and the time complexity of this algorithm is analyzed. E-graphs are then used at the core of various theorem provers and solvers (Detlefs et al., 2005; de Moura and Bjørner, 2008; Barrett et al., 2011). In the 2000s, e-graphs are repurposed for program optimization Tate et al. (2009); Joshi et al. (2006). The technique, known as equality saturation, repeatedly performs non-destructive rewriting on the e-graphs to grow a compact space of equivalent programs. An extraction procedure is then applied to extract the optimal program. In essense, equality saturation mitigates the phase-ordering problem by keeping all programs. This inspires later work on using e-graphs for translation validation (Stepp et al., 2011), floating-point arithematic (Panchekha et al., 2015), semantic code search (Premtoon et al., 2020), and computer-aided design (Wu et al., 2019). However, a generic framework for e-graphs is not yet available, and developing e-graphs–based applications requires implementing e-graphs from scratch and is therefore a tedious effort. Recently, a generic framework for e-graphs, called `egg`, is developed (Willsey et al., 2021). As a result, many projects sprang up building domain-specific projects using e-graphs, including rewrite rule synthesis (Nandi et al., 2021), machine learning compiler (Yang et al., 2021; Smith et al., 2021), relational query optimization (Wang et al., 2020), and digital signal processing vectorization (VanHattum et al., 2021).

The connection between e-graphs and relational databases. is first studied in our earlier work on relational e-matching (Zhang et al., 2022). In relational e-matching, we proposed

to view an e-graph as a relational database, which allows us to make e-matching orders of magnititude faster and prove desired theoretical properties. However, to use this technique, one has to keep both the e-graph and its relational representation and convert back and forth, which limits its practical adoptions. We build on this work, which only takes a static relational snapshot of e-graph each time, and explore how e-graphs as a relational database will behave dynamically. This saves us from the labor of keeping and syncing between the two e-graph representations and further exploits the benefits of the relational e-matching approach.

Many works on improving e-graphs focus on the efficiency and usability of e-graphs. Relational e-graphs bring a new perspective to some of these works. For example, de Moura and Bjørner (2007) studied the incremental maintenance problem of the e-matching procedure and one of its standard extension called multi-patterns. Yang et al. (2021) also proposed an algorithm to extend e-graph frameworks with multi-patterns for equality saturations. In relational e-graphs, multi-patterns are supported naturally (Zhang et al., 2022), and incremental maintenance can be achieved using semi-naïve evaluation, which is a standard technique for evaluating Datalog programs (Balbin and Ramamohanarao, 1987). Many applications extend the expressiveness of e-graphs by writing domain-specific analyses in a general purpose language. For example, to reason about lambda calculus in `egg`, a user may want to implement an analysis that manually tracks the set of free and bound variables (Willsey et al., 2021) in Rust. Such analyses can be written as rules in pure relational e-graphs. For some other works, the problem of finding and understanding its relational dual is still open to future research. For instance, applications like SMT solvers not only want to know if two terms are equivalent, but also why they are equivalent. Techniques are developed to generate proofs for equivalences in e-graphs (Nieuwenhuis and Oliveras, 2005). It is speculated that proofs for congruence closure in relational form may just be database provenance (Green et al., 2007; Zhao et al., 2020). Moreover, scheduling is a critical component of equality saturation (Willsey et al., 2021), and a good scheduling algorithm is a key enabler of scalable equality saturations. However, for relational languages like Datalog, the scheduling problem

is less studied, because Datalog programs are usually run to fixpoint, while the fixpoint for equality saturation is usually infinitary. A future direction is to study the scheduling problem for relational e-graphs.

Other data structures for compact program representation have also been long studied in the literature for decades, in particular version space algebras (VSAs) (Wolfman et al., 2001; Polozov and Gulwani, 2015) and finite tree automata (Wang et al., 2017a,b). Recently, it is shown that both e-graphs and VSAs are special cases of finite tree automata (Koppel, 2021). A natural question is therefore if we can encode VSAs, finite tree automata, and operations over them in a relational approach, and how we can possibly benefit from such an encoding for tasks like program synthesis and program optimization.

4.2 Relational databases and Datalog

Our work is closely related to works on Datalog and relational database. Relational e-graphs are directly inspired by work on data dependencies and the chase. Equality generating dependencies (EGDs) are data dependencies of the form $\forall \vec{x}.\lambda(\vec{x}) \rightarrow x_i = x_j$, which asserts the equality between terms under conditions given by predicate λ. EGDs generalizes the congruences in equality saturation, e.g., the congruence property of binary operator *add* can be written as $\forall x, y, z_1, z_2.add(x, y, z_1), add(x, y, z_2) \rightarrow z_1 = z_2$. Tuple generating dependencies (TGDs) are another kind of data dependencies of the form $\forall \vec{x}, \lambda(\vec{x}) \rightarrow \exists \vec{y}, \rho(\vec{x}, \vec{y})$, which essentially generalizes Datalog rules with existential quantifiers in the body. TGDs generalizes rewrite rules in equality saturation. For example, the associativity is expressed as TGD $\forall x, y, xy, z, xyz.add(x, y, xy), add(xy, z, xyz) \rightarrow \exists yz.add(y, z, yz), add(x, yz, xyz)$. The chase is a family of iterative algorithms for reasoning about data dependencies (Deutsch et al., 2008; Benedikt et al., 2017). Therefore, equality saturation can be effectively viewed as a chase algorithm. As a chase procedure, equality saturation has many interesting properties that are worth further study: While there may be many finite universal models to data dependencies in the chase, in equality saturation, there will be at most one finite universal model, which is the core. Moreover, equality saturation terminates if and only if there is a

finite universal model of the given dependencies, which equality saturation will output, In contrast, the (non-core) chase does not necessarily terminate even when a finite universal solution of the input dependencies exists, or such a finite solution is very expensive to compute (with the core chase (Deutsch et al., 2008)). A future direction is to understand equality saturation using the theories developed in database research.

Many applications using e-graphs rely on domain-specific analyses. To express these analyses in a generic e-graph framework, `egg` proposed a framework called e-class analyses. An e-class analysis can be thought of as an aggregation of information in the corresponding e-class's e-nodes and their children's e-class analyses. E-class analyses are interdependent, because the e-class analysis of one e-class depends on its children's e-class analyses. Therefore, e-class analyses can be formulated as recursive aggregates in Datalog. However, unconstrained aggregates within recursions are dangerous, as it can lead to non-monotonic programs and there may not be (stable) models, so in practice aggregate stratification is encorced. Several approaches are proposed to loosen the aggregate stratification requirement (Ross and Sagiv, 1992; Conway et al., 2012; Abo Khamis et al., 2022b; Green et al., 2007; Madsen et al., 2016). Among these approaches, the e-class analyses naturally matches the lattice semantics proposed by Flix (Madsen et al., 2016), as each e-class analysis monotonically maintains a lattice, which is associated with each e-class, In Flix, relations are optionally annotated with a lattice, which aggregates over values passed it according to the lattice join operation. Our relational e-graphs extends Flix by allowing multiple atoms in the head. A similar lattice-based approach is studied in the Bloom$^{\mathrm{L}}$ language (Conway et al., 2012).

Following relational e-matching (Zhang et al., 2022), we use worst-case optimal join for solving queries over relational e-graphs. Different from relational e-matching, which only needs to consider static snapshots of a relational database (i.e., no writes), we have to also consider insertions and updates to the database. Several previous works considered adopting worst-case optimal joins for practical systems. Two critical dimensions in the design space is the design of indexes and query planning. For indices, EmptyHeaded only

considers scenarios with static dataset (Aberger et al., 2017) and therefore their indexes are read-optimized and need to be precomputed. Our context for using worst-case optimal join requires to update the database and is therefore more similar to LogicBlox. LogicBlox is a commercial database system that uses leapfrog triejoin (LFTJ) (Veldhuizen, 2014), which is worst-case optimal, for its query processing (Aref et al., 2015). To support both LFTJ and efficient updates, LogicBlox uses write-optimizied B-trees for indexes. Different indexes are created and maintained when they are used by query plans generated from the query optimizer. Recently, Freitag et al. (2020) implements a worst-case optimal join inside the Umbra database system. In their design, the indices are built on-the-fly during join processing and are optimized for fast building. This design allows efficient updates to databases since no additional indices need to be maintained during updates.

In terms of query optimization, LogicBlox uses a sampling-based technique to pick a good query plan. EmptyHeaded uses generalized hypertree decomposition (GHD), which allows it to provide guarantees even stronger than those provided by standard worst-case optimal joins. (Freitag et al., 2020) uses cardinality information to optimize its query plans, and allows hybrid query plans that use binary joins like hash joins when it is deemed that worst-case optimal join does not offer a benefit. Currently, the query optimizer of our relational e-graph is relatively simple, and we hope to study and adopt techniques used in other database systems that use worst-case optimal joins.

Chapter 5

CONCLUSION

In previous chapters, we study a non-relational optimization to e-matching and its properties. Despite being able to optimize some queries, the non-relational e-matching is limited by the graph representation of e-graphs and many optimizations that are easily applicable to relational e-matching are not doable in this non-relational approach. However, the relational e-matching approach only concerns the e-matching procedure and does not tell us how to perform other operations over an e-graph relationally, Therefore, we turn our focus to Datalog, an expressive relational language, and study approaches to encoding e-graphs in existing Datalog systems. We present encodings in two Datalog systems, namely Soufflé and Rel. The result is mixed: although we can express e-graphs in both systems declaratively, they are much slower than a mature e-graph framework like egg, partly because egg is well optimized for congruence reasoning workload and partly because of many limitations of the Datalog language. We therefore propose a new relational language based on Datalog with first-class congruence reasoning. We give an introduction to the language prototype and described its evaluation algorithm and model semantics, and a system for the proposed language is in active development.

Although we have settled on the idea of building a relational language with first-class support of congruence, there are still more questions awaiting future work.

What will scheduling and proofs look like for this relational language? Scheduling is less concerned for traditional Datalog systems, since Datalog rules are usually run to saturation, yet it is critical to the performance of egg. A good scheduling algorithm can make egg to discover desired facts order of magnititude faster, and a naïve scheduling algorithm may spend the majority of its time populating facts that are hardly useful. In egg, rule scheduling

is relatively easy because they only need to consider equational rewrite rules. However, for relational e-graphs, their language is a superset of Datalog, which makes rule scheduling more complicated. More importantly, e-class analyses, which are usually expressed as custom Rust code, are now also expressed as rewrite rules. However, different from these equational rewrite rules for "growing" an e-graph, which usually are non-terminating, e-class analyses are usually run to saturation (even though there are cases e-class analyses can be non-terminating). Rule scheduling therefore may need to distinguish between different kinds of rules and schedule differently. Proofs are in a similar situation: although algorithms have been devised for fast and quality proof generation, they work exclusively over e-graph operations. For them to work for relational e-graphs, it is necessary to study situations beyond e-class merges. The research question here is essentially proof generation for data dependencies (i.e., tuple-generating depednencies and equality generating dependencies).

What are the theoretical connections between relational e-graphs and well-established procedures like SMT solving and the chase? Chapter 4 hinted as the possible connections between relational e-graphs and the chase. This allows us to study relational e-graphs from a theoretical perspective using techniques established in the chase. Moreover, e-graphs bear many connections with theory solving and are originally proposed to solve the theoy of equalities. Relational e-graphs make one step further to also allow predicates to be expressed as relations. This makes our surface language strikingly similar to the textual language SMT solvers use. One key difference is that SMT solvers support backtracking and therefore disjunction, which relational e-graphs do not support. In fact, a key characteristic of both traditional e-graphs and relational e-graphs is their monotonicity. Yet the question remains what other potential semantic differeces are or if we are essentially building an efficient non-disjunctive SMT solver.

What are the potential uses of relational e-graphs One of the motivation for relational e-graphs is the lack of expressivity in traditional e-graphs beyond equational reasoning. Relational e-graphs supplement traditional e-graphs with the capability to express non-equational reasoning. The question is therefore what new applications this new expressive power enables

the developers to build. Currently, we are working on using relational e-graphs to write type inference algorithms for Hindley-Milner type systems, to write solvers for algebraic equations, and to write interval analyses for arithematic tools like Herbie. A future direction is to find new problem domains where egglog can be used—we have built the correct hammer, but where are the nails?

How should semi-naïve evaluation be implemented for this language and does it provide significant speedup? Semi-naïve evalaution is well-known in the Datalog literature, and a similar technique for e-graph is proposed in literature called incremental e-matching. While incremental e-matching is complicated and does not necessarily work for the intensive update scenarios of equality saturation, semi-naïve may just work. However, as we have not implemented semi-naïve evaluation yet, it is unclear what it takes to implement it and to what extent it can bring speedup, if any.

How does the performance of relational e-graphs compare to that of traditional e-graphs? Finally, we have not benchmarked the new relational e-graphs we are building against traditional e-graphs. My hypothesis is that theoretically relational e-graphs should have similar or even better performance compared to traditional e-graphs, since the indices a traditional e-graph rely on can be easily expressed relationally. However, to achieve this practically requires a fair amount of engineering efforts.

All these questions have not been fully answered yet. However, I believe our design of egglog and the accompanying egg-smol[1] implementation are the first steps to answering these important questions.

[1]https://github.com/mwillsey/egg-smol.

BIBLIOGRAPHY

Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246

Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022a. Convergence of Datalog over (Pre-) Semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/3517804.3524140

Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022b. Convergence of Datalog over (Pre-) Semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/3517804.3524140

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: A Tale of Inter-Engine Portability for Datalog-Based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference*

on Management of Data (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, USA, 739–748. https://doi.org/10.1109/FOCS.2008.43

I. Balbin and K. Ramamohanarao. 1987. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.* 4, 3 (sep 1987), 259–262. https://doi.org/10.1016/0743-1066(87)90004-5

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the Chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 37–52. https://doi.org/10.1145/3034786.3034796

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*. 1. https://doi.org/10.1145/2391229.2391230

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Au-*

tomated Deduction – CADE-21, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

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 Berlin Heidelberg, Berlin, Heidelberg, 337–340.

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

Alin Deutsch, Alan Nash, and Jeff Remmel. 2008. The Chase Revisited. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Vancouver, Canada) *(PODS '08)*. Association for Computing Machinery, New York, NY, USA, 149–158. https://doi.org/10.1145/1376916.1376938

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (oct 1980), 758–771. https://doi.org/10.1145/322217.322228

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

Thom Frühwirth. 2009. *Constraint handling rules*. Cambridge University Press.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An Efficient Interpreter for Datalog by De-Specializing Relations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 681–695. https://doi.org/10.1145/3453483.3454070

Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A Practical Algorithm for Generating Optimal Code. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 967–989. https://doi.org/10.1145/1186632.1186633

George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An Efficient Data Structure for Must-Alias Analysis. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 48–58. https://doi.org/10.1145/3178372.3179519

James Koppel. 2021. Version Space Algebras are Acyclic Tree Automata. https://doi.org/10.48550/ARXIV.2107.12568

Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

Gerhard Köstler, Werner Kiessling, Helmut Thöne, and Ulrich Güntzer. 1995. Fixpoint

Iteration with Subsumption in Deductive Databases. *J. Intell. Inf. Syst.* 4, 2 (mar 1995), 123–148. https://doi.org/10.1007/BF00961871

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford, CA, USA. AAI8011683.

Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/3196959.3196990

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA'05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Port-

land, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. *SIGPLAN Not.* 50, 10 (oct 2015), 107–126. https://doi.org/10.1145/2858965.2814310

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

Fergal Reid and Martin Harrigan. 2013. *An Analysis of Anonymity in the Bitcoin System*. Springer New York, New York, NY, 197–223. https://doi.org/10.1007/978-1-4614-4139-7_10

Rel documentation team. 2022. References material for the Rel language. https://docs.relational.ai/rel/ref/overview

Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(PODS '92)*. Association for Computing Machinery, New York, NY, USA, 114–126. https://doi.org/10.1145/137097.137852

Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless Deductive Inference via Macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3497776.3517779

Tom Schrijvers and Thom Frühwirth. 2006. Optimal union-find in Constraint Handling

Rules. *Theory and Practice of Logic Programming* 6, 1-2 (2006), 213–224. https://doi.org/10.1017/S1471068405002541

Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Proceedings of the 23rd international conference on Computer Aided Verification* (Snowbird, UT, USA). Springer-Verlag, Berlin, Heidelberg, 737–742. http://www.cs.cornell.edu/~ross/publications/eqsat/

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

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. *Vectorization for Digital Signal Processors via Equality Saturation.* Association for

Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017a. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (dec 2017), 30 pages. https://doi.org/10.1145/3158151

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (dec 2017), 30 pages. https://doi.org/10.1145/3158151

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

Max Willsey. 2020. E-graphs in Julia (Part I). https://docs.rs/egg/latest/egg/tutorials/_01_background/index.html

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Steven Wolfman, Pedro Domingos, and Daniel Weld. 2001. Programming By Demonstration Using Version Space Algebra. *Machine Learning* 53 (12 2001). https://doi.org/10.1023/A:1025671410623

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Trans. Graph.* 38, 6, Article 195 (nov 2019), 14 pages. https://doi.org/10.1145/3355089.3356518

Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf

Yihong Zhang. 2021. *Faster and Worst-Case Optimal E-Matching.* Bachelor's thesis. University of Washington. http://effect.systems/doc/ug-thesis/thesis.pdf

Yihong Zhang. 2022. A Trick that Makes Classical E-Matching Faster. http://effect.systems/blog/ematch-trick.html

Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational E-Matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (jan 2022), 22 pages. https://doi.org/10.1145/3498696

David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-Scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 7 (apr 2020), 35 pages. https://doi.org/10.1145/3379446

Philip Zucker. 2020a. E-Graph Pattern Matching (Part II). https://www.philipzucker.com/egraph-2/

Philip Zucker. 2020b. E-graphs in Julia (Part I). https://www.philipzucker.com/egraph-1/

Philip Zucker. 2022a. E-Graphs in Souffle IV: It's actually kind of fast this time. https://www.philipzucker.com/souffle-egg4/

Philip Zucker. 2022b. A Questionable Idea: Hacking findParent into Souffle with User Defined Functors. https://www.philipzucker.com/souffle-functor-hack/