Better Together: Unifying Datalog and Equality Saturation

YIHONG ZHANG, University of Washington, USA
YISU REMY WANG, University of Washington, USA
OLIVER FLATT, University of Washington, USA
DAVID CAO, UC San Diego, USA
PHILIP ZUCKER, Draper Laboratory, USA
ELI ROSENTHAL, Google, USA
ZACHARY TATLOCK, University of Washington, USA
MAX WILLSEY, University of Washington, USA

We present `egglog`, a fixpoint reasoning system that unifies Datalog and equality saturation (EqSat). Like Datalog, it supports efficient incremental execution, cooperating analyses, and lattice-based reasoning. Like EqSat, it supports term rewriting, efficient congruence closure, and extraction of optimized terms.

We identify two recent applications—a unification-based pointer analysis in Datalog and an EqSat-based floating-point term rewriter—that have been hampered by features missing from Datalog but found in EqSat or vice-versa. We evaluate `egglog` by reimplementing those projects in `egglog`. The resulting systems in `egglog` are faster, simpler, and fix bugs found in the original systems.

CCS Concepts: • **Theory of computation** → *Constraint and logic programming*; **Equational logic and rewriting**.

Additional Key Words and Phrases: Program optimization, Rewrite systems, Equality saturation, Datalog

1 INTRODUCTION

Equality saturation (EqSat) and Datalog are both fixpoint reasoning frameworks with many applications, extensions, and high-quality implementations [Jordan et al. 2016; Willsey et al. 2021]. They share a common setup: the user provides *rules* and an initial set of *facts* (a term in EqSat and a database in Datalog), then the system derives a larger and larger set of facts from those inputs. However, their commonalities have not—until now—been fully realized or exploited. As a result, the frameworks have developed independently and are used in different domains. Datalog is well-studied by the databases community, and practitioners use modern implementations to build program analyses [Balatsouras and Smaragdakis 2016; Barrett and Moore 2013; Smaragdakis and Bravenboer 2010]. Equality saturation is a more recent, term-centric technique favored in the programming languages community for program optimization and verification.

As users apply EqSat and Datalog to new, more demanding problems, the limitations of each tool become apparent. For example, Herbie [Panchekha et al. 2015], a tool that uses EqSat to optimize floating-point accuracy, relies on unsound rewrites because it lacks the analyses to prove that certain rewrites are safe (e.g. $x/x \longrightarrow 1$ only if $x \neq 0$). To combat the unsoundness, Herbie must validate the results of EqSat and discard them if unsoundness was detected. On the Datalog side, `cclyzer++` [Barrett and Moore 2013], a recent points-to analysis system implemented in Datalog that supports Steensgaard analyses [Steensgaard 1996] for LLVM [Lattner and Adve 2004] resorted to an ad-hoc implementation of union-find, because the provided implementation of equivalence relations was too slow. The resulting implementation's complexity led to bugs in the pointer analysis. In short, EqSat struggles to support rich analyses, and equational reasoning in Datalog is complex and slow.

Authors' addresses: Yihong Zhang, University of Washington, USA; Yisu Remy Wang, University of Washington, USA; Oliver Flatt, University of Washington, USA; David Cao, UC San Diego, USA; Philip Zucker, Draper Laboratory, USA; Eli Rosenthal, Google, USA; Zachary Tatlock, University of Washington, USA; Max Willsey, University of Washington, USA.

Our key insight is that the efficient equational reasoning of EqSat and the rich, composable semantic analyses of Datalog make up for each other's weaknesses, and unifying the two paradigms brings together—and goes beyond—the best of both worlds. In fact, spontaneous developments in both communities have already begun converging towards each other: Datalog tools have added efficient equivalence relations [Nappa et al. 2019], lattices [Madsen et al. 2016; Sahebolamri et al. 2022], and some support for datatypes [Developers [n.d.]], while the EqSat community has recently developed support for conditional rewriting, lattice-based analyses [Cheli 2021; Willsey et al. 2021], and relational pattern matching [Zhang et al. 2022]. We bring this trend to completion and close the gap between EqSat and Datalog.

In this work, we propose `egglog`, a fixpoint reasoning system that subsumes both EqSat and Datalog. It contains all of the innovations listed above as well as new ones, and it addresses crucial limitations that have prevented progress in real-world applications. `egglog` is essentially a Datalog engine with two main extensions. First, `egglog` has a built-in, extensible notion of equality. The user can assert that two terms are equivalent, from which point on they are indistinguishable to the system. For example, consider a relation with a single tuple: $R = \{(a, b)\}$ for distinct a and b. The query $R(x, x)$ would yield nothing, but if the user asserts that a and b are equivalent, then the query would return the equivalence class containing both a and b. Second, `egglog` has built-in support for (uninterpreted) functions. From a relational perspective, a function is a relation with a functional dependency from its arguments to its output, i.e., the output is uniquely determined by the arguments. However, user-extensible equality introduces challenges for maintaining functional dependencies. Consider a function f such that $f(a) = b$ and $f(c) = d$, but b does not (yet) equal d. What happens when the user asserts that a and c are equivalent? An `egglog` function can be annotated with a *merge expression*, a novel mechanism that `egglog` uses to resolve functional dependency violations by combining the two conflicting output values. In the above case, f's merge expression might assert that $b = d$ (essentially asserting congruence of f), or return the supremum of b and d. The flexibility of merge expressions allows `egglog` to exceed the expressive power of both EqSat and Datalog extensions with lattices. The high-level `egglog` language allows the user to specify complex interactions among terms, equivalence classes, and lattice values. At the same time, highly optimized algorithms for relational and equational reasoning work together to make `egglog` efficient.

The combination of EqSat and Datalog also brings many practical—if somewhat more prosaic—benefits. For example, Zhang et al. [2022] observed that EqSat is hampered by inefficient e-matching (pattern matching modulo equality) algorithms, and that a relational approach can be vastly more efficient. `egglog`'s Datalog-first design naturally supports efficient e-matching by reducing it to a relational query. This goes even further: incremental e-matching is only supported in some SMT solvers like Z3 [De Moura and Bjørner 2008] and has not yet made its way into EqSat implementations, while `egglog` supports them *for free* with semi-naïve evaluation [Balbin and Ramamohanarao 1987], a common technique that makes Datalog incremental. `egglog`'s support for functions provides the basis for working with terms, which only have limited support in other Datalog systems [Developers [n.d.]]. Users can also define multiple functions and datatypes to model their domain, unlike most EqSat tools [Cheli 2021; Willsey et al. 2021] where users are forced to use a single, ad-hoc datatype. Finally, `egglog` is designed as a language (as well as a library), making it more accessible than EqSat libraries [Cheli 2021; Willsey et al. 2021] that are locked to their implementation language.

We perform two case studies showing that `egglog` out-performs state-of-the-art applications of EqSat and Datalog respectively. First, we show that `egglog` makes Steensgaard-style points-to analyses faster and easier to write. Compared to the Soufflé Datalog system, `egglog` computes the points-to analysis 4.96× faster. Second, we demonstrate the power of `egglog` with a new,

$$E(1,2).$$
$$E(2,3).$$
$$E(3,4).$$
$$TC(x,y) \text{ :- } E(x,y).$$
$$TC(x,y) \text{ :- } TC(x,z), E(z,y).$$

(a) Transitive closure in Datalog. Facts (e.g. $E(1,2)$) are given as rules without bodies.

iter#	E	TC
0	$\emptyset$	$\emptyset$
1	$\{(1,2),(2,3),(3,4)\}$	$\emptyset$
2	$\{\ldots\}$	$\{(1,2),(2,3),(3,4)\}$
3	$\{\ldots\}$	$\{\ldots,(1,3),(2,4)\}$
4	$\{\ldots\}$	$\{\ldots,(1,4)\}$

(b) Execution trace of transitive closure. "..." includes tuples from the cell directly above.

Fig. 1. Transitive closure is the classic Datalog example. It iteratively computes the transitive closure (TC) of an edge relation E by applying the rules in Figure 1a.

sound implementation of Herbie's EqSat procedure. This allows Herbie to perform aggressive optimizations soundly, and return results faster given the same error tolerance.

In summary, this paper makes the following contributions:

- We introduce a bottom-up, Datalog-like logic language for equality saturation and similar unification-based algorithms.
- We present a fixpoint semantics for the core language of `egglog`.
- We present an implementation for `egglog` with optimizations from database research such as semi-naïve evaluation.

2 BACKGROUND

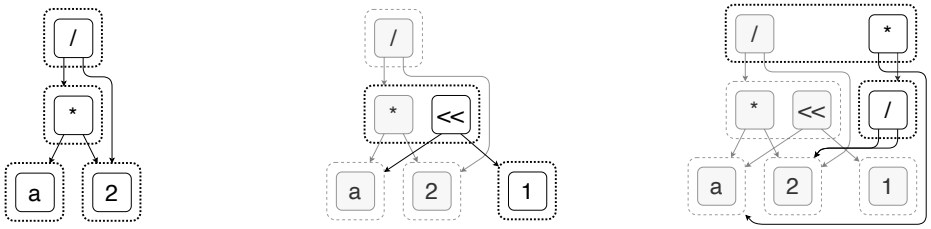
`egglog` is designed as a Datalog variant with extensions that make it subsume EqSat. This section will introduce both Datalog and EqSat in their own terms, while Section 3 will show how they both fit within the `egglog` framework.

2.1 Datalog

Figure 1 shows a Datalog program to compute the transitive closure of a graph. Datalog is a recursive database query language that represents data as *relations*. Each relation is a set of tuples, and all tuples in the same relation share the same arity. A Datalog program consists of a set of *rules*. Each rule is a *conjunctive query* of the form $Q(\mathbf{x}) \text{ :- } R_1(\mathbf{x}_1), R_2(\mathbf{x}_2), \ldots, R_n(\mathbf{x}_n)$ where each $\mathbf{x}$ and $\mathbf{x}_i$ is a tuple of variables or constants. The atom $Q(\mathbf{x})$ is called the *head* of the rule, and the atoms $R_i(\mathbf{x}_i)$ comprise the *body*. The body binds variables to be used in the head to create new facts; all variables in the head must appear in the body. Specifically, running a rule adds the following facts: $\{Q(\mathbf{x}[\sigma]) \mid \bigwedge_i \mathbf{x}_i[\sigma] \in R_i\}$, where σ is a substitution that maps all the variables in the rule to constants. In other words, querying the body creates substitutions such that every substituted body atom is in the database; these substitutions are then applied to the head to create new facts.

Each rule can be seen as a function from the current database to a new database that includes the facts created by the rule; call this function T_r for some rule r. The set of all rules r in a Datalog program p therefore defines a function T_p from the current database to a new database: $T_p(\text{DB}) = \bigcup_{r \in p} T_r(\text{DB})$. This function is called the immediate consequence operator (ICO) of the program, which we denote T_p. To run a Datalog program, we start with an empty database and repeatedly apply T_p until the database stops changing. A fundamental result in Datalog is that every program terminates, and the final result is the least fixpoint of T_p [Abiteboul et al. 1995].

Datalog became popular in programming languages research as a declarative language for specifying large-scale program analyses such as points-to analyses [Smaragdakis and Bravenboer



(a) E-graph represents $(a \times 2)/2$. (b) Rewrite $x \times 2 \rightarrow x \ll 1$. (c) Rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

Fig. 2. Applying rewrites over an example e-graph (figures from Willsey et al. [2021]). A solid box denotes an e-node, and a dotted box denotes an e-class. E-nodes consist of a function symbol and children e-classes, and e-classes contain a set of e-nodes.

2010]. In order to support abstract interpretation-style analyses, researchers have extended Datalog to work over lattices. In the lattice semantics, a relation is viewed as a function from tuples to a lattice, We then generalize Datalog rules to be over functions: $Q(\mathbf{x}) \mapsto x$:- $R_1(\mathbf{x_1}) \mapsto x_1, \cdots, R_n(\mathbf{x_n}) \mapsto x_n$. The value of Q on input $\mathbf{x}$ is the supremum of valid xs producible by the body, i.e., $Q(\mathbf{x}) = \bigsqcup\{x \mid \bigvee_{\mathbf{x}_{\text{free}}} R_1(\mathbf{x_1}) = x_1 \wedge \ldots \wedge R_n(\mathbf{x_n}) = x_n\}$ where $\mathbf{x}_{\text{free}}$ is the set of variables in the body that do not appear in the head and $\sqcup$ is a lattice join (i.e., supremum) operator. `egglog`'s support for lattices is motivated by other modern Datalog implementations [Abo Khamis et al. 2022; Madsen et al. 2016; Sahebolamri et al. 2022] that support this extension.

2.2 Equality Saturation

Traditional term rewriting applies one rule at a time and forgets the original term after each step, so it is sensitive to the ordering of the rewrites. For example, rewriting $(a \times 2)/2$ to $(a \ll 1)/2$ is locally good, but it prevents future opportunities to cancel out $2/2$. Equality saturation (EqSat) [Tate et al. 2009] is a technique to mitigate this phase-ordering problem. EqSat fires all the rules in each iteration and keeps both original and rewritten terms in a special data structure called the e-graph. An e-graph [Nelson 1980] is a compact data structure that represents large sets of terms efficiently. An e-graph is a set of *e-classes*, and each e-class is a set of equivalent *e-nodes*. An e-node is function symbol with children e-classes (not e-nodes).

An e-graph can compactly represent an exponential number of terms compared to the size of the e-graph. We say an e-graph *represents* a term t if any of its e-classes represents t, and an e-class represents t if any e-node in the e-class represents it. An e-node $t = f(c_1, \ldots, c_n)$ represents a term $f(t_1, \ldots, t_n)$ if each c_i represents t_i. An e-graph induces an equivalence relation over terms: two terms are considered equivalent if they are represented by the same e-class. This equivalence relation is also congruent: if an e-graph represents two terms $a = f(a_1, \ldots, a_n)$ and $b = f(b_1, \ldots, b_n)$ such that e-graph shows $a_i \equiv b_i$, then the e-graph can also show $a \equiv b$.[1]

Figure 2 shows an example e-graph and two rule applications. We start with the initial e-graph representing only term $(a \times 2)/2$. To apply a rewrite rule $x \times 2 \rightarrow x \ll 1$, we first search for matches of left-hand patterns using a procedure called *e-matching* (pattern matching modulo equality). This produces substitutions (in this case, only one: $\{x \mapsto a\}$) that we then we apply to right-hand side pattern. Each resulting term (e.g., $a \times 2$) is finally merged into the e-class that the left-hand side pattern matched.

[1] In e-graph implementations that canonicalize e-nodes, congruence amounts to deduplication of e-nodes since nodes a and b would canonicalize to identical e-nodes.

Extensions. Standard EqSat is purely syntactic. In some cases, this prevents users from writing sound rewrites. For example, the rewrite $\sqrt{x^2} \rightarrow x$ is sound iff x is non-negative, but proving this requires semantic analyses. A recent technique called e-class analyses [Willsey et al. 2021] allows for semantic analyses in EqSat. An e-class analysis associates every e-class in an e-graph with a semi-lattice value that is a semantic abstraction of the term. During the EqSat algorithm, the lattice data are propagated from children to parents e-classes, and merged via lattice joins. For example, an analysis could track the lower bounds of e-classes, which are initially $-\infty$ and increase over time as new terms are represented in these e-classes via rewrites. In egg, the most popular EqSat toolchain, e-class analyses are currently limited. An e-graph can only have a single e-class analysis, it can only propagate information *upwards* from children to parents, and it requires writing low-level code in the host programming language (Rust in egg's case).

Multi-patterns are another commonly used extension to e-matching (and thus EqSat). Typically, e-matching only supports patterns matching a single term each. A multi-pattern is a set of multiple patterns to be matched simultaneously. For example, TenSat [Yang et al. 2021] is an equality saturation based tensor graph optimizer that uses rewrite rules to share matrix multiplications. It matches patterns $e_1 = matmul(M_1, M_2)$ and $e_2 = matmul(M_1, M_3)$ simultaneously, and then creates the expression $e_3 = matmul(M_1, concat(M_2, M_3))$ and merges e_1 with $split_1(e_3)$ and e_2 with $split_2(e_3)$. Previous works have developed algorithms for multi-patterns [de Moura and Bjørner 2007; Yang et al. 2021], but they are suboptimal and complex.

Relational e-matching [Zhang et al. 2022] is a recent technique to improve e-matching performance, including on multi-patterns, by reducing it to a relational query. However, relational e-matching suffers from the "dual representation" problem. An equality saturation engine has to switch back and forth between the e-graphs and the relational database representations. This can sometimes take a significant amount of the run time, reducing the benefits of this approach. Relational e-matching hints at the fundamental connection between e-graphs and relational databases, but it only applies the insights to e-matching. We further exploit the connection in `egglog`, building a Datalog-inspired system that captures the entire EqSat algorithm and goes beyond.

3 EGGLOG

`egglog` is a logic programming language that bears many similarities to Datalog, and it also incorporates features that allow for program optimization and verification as in equality saturation. In this section, we approach `egglog` by example, starting from the Datalog perspective and adding features until it subsumes equality saturation.

3.1 Datalog in egglog

`egglog` uses a concrete syntax based on s-expressions, but despite this surface-level difference, readers familiar with Datalog should find many `egglog` programs familiar. The program in Figure 3a computes the transitive closure of a graph, just like the Datalog program in Figure 1a. It first declares two relations of pairs of 64-bit integers (`i64` is one of `egglog`'s base types). The `edge` relation stores the edges of a graph and is initially populated manually on lines 9-11. The `path` relation is populated by the rules on lines 4-7. These rules compute the transitive closure of the `edge` relation. Finally, the last two lines execute the program and check that there is a path from 1 to 4.

Let us take a closer look at the second rule in Figure 3a: it states that if there is a path from x to y and an edge from y to z, then there is a path from x to z. In `egglog`, a rule has two parts: a *query* and a list of *actions*.[2] The query is a set of *patterns*, all of which must match for the rule to fire. If

[2]Note that this is backwards from the more traditional Datalog syntax: `path(X, Z) :- path(X, Y), edge(Y, Z).` An `egglog` rule's query and actions are analogous to the body and head of a Datalog rule, respectively.

```
(relation edge (i64 i64))
(relation path (i64 i64))

(rule ((edge x y))
      ((path x y)))
(rule ((path x y) (edge y z))
      ((path x z)))

(edge 1 2)
(edge 2 3)
(edge 3 4)

(run)
(check (path 1 4)) ;; succeeds
```

(a) Reachability in the classic Datalog style.

```
(function edge (i64 i64) i64)
(function path (i64 i64) i64 :merge (min old new))

(rule ((= (edge x y) len))
      ((set (path x y) len)))
(rule ((= (path x y) xy) (= (edge y z) yz))
      ((set (path x z) (+ xy yz))))

(set (edge 1 2) 10)
(set (edge 2 3) 10)
(set (edge 1 3) 30)

(run)
(check (path 1 3)) ;; prints "20"
```

(b) Reachability including shortest path length.

Fig. 3. egglog supports classic Datalog programs like reachability written in the natural way. Functions and `:merge` allow egglog to support Datalog with lattices similar to tools like Flix [Madsen et al. 2016] or Ascent [Sahebolamri et al. 2022].

all patterns do match, the query binds each variable to a value. The actions dictate what happens when the rule fires, and they can use the variables that are bound by the query. Typically, as in this example, the actions assert new facts to be added to the database.

3.2 Functions and `:merge`

Unlike traditional Datalog, egglog stores data as partial functions rather than relations. A relation in egglog actually desugars to a function whose return type is the built-in unit type. To model a unary relation R, we can use a function f_R to unit such that $f_R(x) = ()$ if $x \in R$ else undefined. While a Datalog program's rules add tuples to relations, egglog's functions become defined for more and more tuples over the course of a program's execution, a concept that we will explore in more detail in Section 4.2. Every user-defined function in egglog is backed by a map (as opposed to a set in in Datalog). Crucially, the map enforces the functional dependency from inputs to outputs. In other words, a function maps each input to a unique output. Throughout this paper, we will use the term "table" to refer to either the backing map of an egglog function or the backing set of a Datalog relation.

Consider the program in Figure 3b that computes the length of shortest path between all nodes. Ignoring the `:merge` declaration for now, the program is substantially similar to the reachability program in Figure 3a, but it uses functions instead of relations. The program defines `edge` and `path` functions to `i64` rather than functions to unit (i.e., relations). The first rule (the base case) in Figure 3b is similar to before: it says that an edge of length len from x to y implies there is a path of (at most) length len from x to y. The query uses `=` to bind the output of the `edge` function to the variable `len`. In the action, we see the `set` construct, which asserts that a function maps some arguments to a given value. The action in the analogous rule in Figure 3a desugars to `(set (path x y) ())`. Note that if the arguments are already mapped to a value, we need to reconcile the old value with the new one to preserve the functional dependency. This is resolved by the `:merge` declaration which we will describe next.

The second rule in Figure 3b is the transitive case, and here we see the purpose of the `:merge` declaration. This rule says that if there is a path from x to y of length xy, and an edge from y to z of

```
(sort Node)
(function mk (i64) Node)
(relation edge (Node Node))
(relation path (Node Node))

(rule ((edge x y))
      ((path x y)))
(rule ((path x y) (edge y z))
      ((path x z)))

(edge (mk 1) (mk 2))
(edge (mk 2) (mk 3))
(edge (mk 5) (mk 6))

(union (mk 3) (mk 5))
(run)
(check (edge (mk 3) (mk 6)))
(check (path (mk 1) (mk 6)))
```

(a) Combining nodes with unification

```
(datatype Math
  (Num i64)
  (Var String)
  (Add Math Math)
  (Mul Math Math))

;; expr1 = 2 * (x + 3)
(define expr1 (Mul (Num 2) (Add (Var "x") (Num 3))))
;; expr2 = 6 + 2 * x
(define expr2 (Add (Num 6) (Mul (Num 2) (Var "x"))))

(rewrite (Add a b)         (Add b a))
(rewrite (Mul a (Add b c)) (Add (Mul a b) (Mul a c)))
(rewrite (Add (Num a) (Num b)) (Num (+ a b)))
(rewrite (Mul (Num a) (Num b)) (Num (* a b)))

(run)
(check (= expr1 expr2))
```

(b) Basic equality saturation

Fig. 4. Unification and EqSat in egglog.

length yz, then there is a path from x to z of length $xy + yz$. But what if the function `path` is already defined on the arguments x and z? Functions must map equivalent arguments to unique output, so the **`:merge`** declaration tells egglog how to resolve this conflict. Given the facts later in Figure 3b, the program will discover two paths from 1 to 3: the single edge with length 30 will be discovered first, and then two-edge path with length 20. When `(set (path 1 3) (+ 10 10))` is executed, egglog must come up with a single value to map `(path 1 3)` to. To do this, it evaluates the expression given after **`:merge`** in the function's declaration with **`old`** and **`new`** bound to the old and new values, respectively. In this case, `path`'s **`:merge`** expression simply takes the minimum of the two path lengths. It can be viewed as the join operator, which takes the supremum of a set of values, of the min lattice over `i64` where the partial order is $x \sqsubseteq y \iff x \geq y$. This is similar to the lattice semantics of Flix [Madsen et al. 2016], which also enforces functional dependency by taking the join over the old and new values in some lattice. However, egglog does not restrict the **`:merge`** expression to only join operations over lattices. In the following sections we will show how a **`:merge`** expression that unifies values naturally gives rise to equality saturation.

3.3 Sorts and Equality

egglog gives the user the ability to declare new *uninterpreted* sorts, and functions use these new sorts as inputs or outputs. Crucially, values of user-defined sorts (as opposed to base types) can be *unified* by the **`union`** action. **`union`**-ing two values makes them point to the same element in the underlying universe of uninterpreted sorts. In other words, values that have been unified are essentially indistinguishable to egglog, and all unified variables can be substituted for the same pattern variable.

Consider an enhanced version of path reachability in Figure 4a, where we use unification to implement node contraction (sometimes called vertex contraction). The program declares a new sort `Node`, which is necessary because base types (like `i64`) cannot be unified. The `mk` function is the sole constructor of `Node`s. After the rule declarations (same as in Figure 3a) and some edge

assertions, we see our first `union` action, which takes two arguments of the same user-defined sort and unifies them. Now that nodes 3 and 5 are unified, running the rules will indeed find a path from 1 to 6, a path that did not exist before the unification.

In egglog, users define uninterpreted sorts. A sort is a set of opaque integer values called *ids* and an equivalence relation over those ids. The equivalence relation is implemented with a union-find data structure [Tarjan 1975] that can *canonicalize* ids; two ids are equivalent iff they canonicalize to the same id. Equivalent ids are considered indistinguishable by egglog. In fact, egglog ensures that all ids appearing in the database are canonical. These ids corresponds to e-class ids from the EqSat perspective.

The second line of Figure 4a declares `mk`, a function from `i64` to `Node`. This looks like a constructor, for `Node`s, but it is just like any other function from egglog's perspective; the `mk` function is backed by a map from `i64`s to `Node` ids. In this program, we never query over the `mk` function, but we do call it, treating it like a total function. What is the value of `(mk 1)`, especially since we did not `set` it to anything prior to calling it? Functions in egglog can be imbued with a `:default` expression that extends the partial function as defined by the underlying map to be total. Calling a function `(f x)` will first see if the map for function `f` defines an output for `x`. If so, it returns that output. Otherwise, egglog evaluates the `:default` expression, stores the result in the map, and returns it. Unless otherwise specified, the `:default` for functions that output a user-defined sort is to create an equivalence class in the union-find and return its id (the "make-set" operation); for base types the default `:default` is to crash the program. In other words, calling a function that outputs a user-defined sort is essentially a "get or make-set" operation.

The upcoming subsection will discuss how these features enable equality saturation, but Section 6.1 will demonstrate how the canonicalizing union-find is useful even in a domain where Datalog is traditionally strong: pointer analysis.

3.4 Terms and Equality Saturation

In Figure 4a, the `Node` sort only has a single constructor, `mk`, which takes an `i64`. egglog also supports functions that take user-defined sorts as inputs. In this way, terms are easily constructed in egglog. Combined with the built-in equivalence relation, this term representation directly supports equality saturation in egglog.

Consider Figure 4b, where we define a datatype `Math` that represents a simple language of arithmetic expressions. The `datatype` construct is sugar for a `sort` declaration and a `function` declaration for each constructor. Each constructor is a function that returns a value of type `Math`, and its `:default` behavior creates a fresh id as described above (we will get to its `:merge` behavior shortly). Now we can create terms by just nesting function calls. The `define` statements do just that, creating two terms that we will later prove are equivalent. These statements actually create nullary (constant) functions; `(define x e)` desugars to `(function x () T) (set (x) e)` where `T` is the type of `e`. Evaluating these terms adds them to the database (if not already present) thanks to the `:default` behavior of the constructors.

Term rewriting in equality saturation has two important qualities: (1) pattern matching is done *modulo equality* and (2) rewriting is *non-destructive*, i.e., it only adds information to the e-graph/database. egglog meets both of these criteria: (1) all queries are performed modulo equality since egglog canonicalizes the database with respect to its built-in equivalence relation, and (2) egglog rules (like standard Datalog rules) only add information to the database. egglog provides the `rewrite` statement to simplify creating equality saturation rewrite rules. A `(rewrite p1 p2)` statement desugars to a `rule` that queries for `p1`, binds it to some variable, and `union`s the variable with `p2`: `(rule ((= __var p1)) ((union __var p2)))`.

The program in Figure 4b proves `expr1` equivalent to `expr2` using two uninterpreted rewrites and two that interpret the `Add` and `Mul` functions using the built-in `+` and `*` functions over `i64`. EqSat frameworks like egg require the user to separate the uninterpreted rules from the interpreted part from the computed part using an e-class analysis [Willsey et al. 2021]. egglog uses rules for both.

Like other functions that output user-defined sorts, the `Math` constructors' **`:merge`** behavior is to union the two ids. Combined with egglog's canonicalization, this means that the built-in equivalence relation is also a congruence relation with respect to these functions. Consider the following map for the `Add` function: $\{(a, b) \mapsto c, (a, d) \mapsto e\}$. If we **`union`** b and d such that b is now canonical, canonicalizing the database reveals a violation of the functional dependency from `Add`'s inputs to its output: $(a, b) \mapsto c$ but also $(a, b) \mapsto e$. To resolve the conflict, egglog invokes the **`:merge`** expression of the `Add` function, which in this case **`union`**s c and e.

After running the rules, the final line **`check`**s that `expr1` and `expr2` are now equivalent. The type of both `expr1` and `expr2` is `Math`—a user-defined sort—so the underlying value of the expressions are both ids. Since egglog canonicalizes the database, the **`check`** is implemented with simple equality on the ids. egglog supports optimization as well as verification; the **`extract`** command prints the smallest term equivalent to its given input.

3.5 Beyond EqSat

egglog is not limited to just Datalog or EqSat; the combination allows for possibilities outside the reach of either tool. The combining nodes example from Figure 3a hints at the power of unification in Datalog, and Section 6.1 takes this further by implementing a unification-based pointer analysis in egglog. In Section 6.2, we go the other way, implementing several Datalog-like anaylses to assist an EqSat-powered term rewriting system.

But egglog goes beyond these applications; we describes more egglog pearls in the full version [Zhang et al. 2023], including functional programming, type analyses for the simply typed lambda calculus in equality saturation, type inference for Hindley-Milner type systems, multivariable equational solving, and matrix algebra optimization with Kronecker products. These pearls hint at the potential novel applications in program optimizations and analyses using egglog in the future.

4 SEMANTICS OF EGGLOG

In this section we describe the semantics of core egglog. Core egglog differs from the full egglog language in several aspects. For example, egglog allows multiple actions in a rule while core egglog allows only one atom in the head, and core egglog does not have the **`union`** operation. These egglog features can be desugared into the core language. However, there are also some assumptions we made about the core egglog. For example, we assume the **`:merge`** expression over ids are **`union`** and the **`:merge`** expressions over interpreted constants is the join operator of a given lattice, while egglog allows **`:merge`** to be any valid egglog expression. In other words, the core egglog captures a well-behaving subset of the full egglog language.

4.1 Syntax

Given the set of (interpreted) constants C, the syntax of the core egglog language is shown in Figure 5. An egglog program is defined as a list of rules, and each rule consists of an atom in the head and a list of atoms in the body. An atom has the form $f(p_1, \ldots, p_k) \mapsto o$ and intuitively means function f has value o on $p_1, \ldots, p_k$. A pattern p is a nested expression constructed using function symbols, variables, and constants. We additionally define a ground term (or term) t to be a pattern with no variables, and a ground atom to be an atom where all the patterns are ground terms.

Program	P	::=	$R_1, \ldots, R_n$
Rule	R	::=	A :- $A_1, \ldots A_m$.
Atom	A	::=	$f(p_1, \ldots, p_k) \mapsto o \mid f(p_1, \ldots, p_k)$
Pattern	p	::=	$f(p_1, \ldots, p_k) \mid o$
Term	t	::=	$f(t_1, \ldots, t_k) \mid v$
Base pattern	o	::=	$v \mid x$
Constant	v	::=	$c \mid n$
Interpreted Constant	$c \in C$		
Uninterpreted Constant	$n \in N$		
Variable	$x, y, \ldots$		

Fig. 5. Syntax of core egglog.

A valid egglog program should not explicitly refer to a specific uninterpreted constant n. We include uninterpreted constants in the syntax nonetheless since they are useful when describing the semantics of egglog programs.

4.2 Semantics

Given an infinite set of uninterpreted constants[3] $N = \{n_1, n_2, \ldots\}$ and a complete lattice $L = (C, \sqsubseteq, \sqcup)$ over domain of interpreted constants C, We define $\bot$ to be the least element of L. A schema S is a collection of function symbols and their function signatures, where the types range over $\{N, C\}$. Given a schema S, an instance of S is defined $I = (DB, \equiv)$, where DB is a set of function entries $f(v_1, \ldots, v_k) \mapsto v$ that is consistent with the schema and $\equiv$ is an equivalence relation over $N \cup C$ satisfying $\forall c_1, c_2 \in C. c_1 \equiv c_2 \rightarrow c_1 = c_2$ (i.e., interpreted constants are only equivalent to themselves). For convenience, we also lift set operator (e.g., union, difference) to be between an instance and a database, which applies the operator to instance's database.

Given an arbitrary total order $<$ over $N \cup C$, we define the canonicalization function $\lambda_{\equiv}(t) = \min t' : t' \equiv t$. For convenience, we lift $\lambda_{\equiv}$ to also work on sets and the whole database instances by pointwise canonicalization.

Before proceeding to define the semantics of an egglog program, we need to first define what it means for a ground atom (an atom without variables) to be in the database and what it means to add one to the database. First, we use the judgement $I \vdash A$ to denote a ground atom is contained in the database I.

$$\frac{I = (DB, \equiv) \qquad \begin{array}{c} I \vdash t_i \mapsto v_i \text{ for } i = 1 \ldots k \\ f(v_1, \ldots, v_k) \mapsto v \in DB \end{array}}{I \vdash f(t_1, \ldots, t_k) \mapsto v} \qquad \frac{}{I \vdash v \mapsto v} \qquad \frac{I \vdash f(t_1, \ldots, t_k) \mapsto v \text{ for some } v}{I \vdash f(t_1, \ldots, t_k)}$$

We also define $\mathit{flatten}_I$ in Figure 6 to flatten function entries to be inserted into I given a nested ground atom A.

Now we can define the semantics of an egglog program. It consists of two parts: the immediate consequence operator and the rebuilding operator. We can define the (inflationary) immediate consequence operator $T_P^{\uparrow}$ as follows.[4] Let σ denote a substitution that maps variables to constants,

[3]These uninterpreted constants play a similar role as e-class ids in EqSat or labelled nulls in the chase from the database literature.

[4]The definition of immediate consequence operator in standard Datalog does not union with DB, because rule applications in standard Datalog are monotone. This is not the case in egglog in general. For example, rule $Q(e)$:- $lo(e) \mapsto 5$, where lo tracks the lower bound of an expression, is not monotone because the value of $lo(e)$ can increase over time. Although one can adapt the meet semantics of Flix [Madsen et al. 2016] for relational joins to enforce monotonicity, we do not do this in

$$\mathit{flatten}_I(A) = s \quad \text{where } (v, s) = \mathit{aux}(A)$$

$$\mathit{aux}(f(t_1, \ldots, t_k) \mapsto v) = \left(v, \{f(v_1, \ldots, v_k) \mapsto v\} \cup \bigcup_{i=1,\ldots,k} s_i\right)$$
$$\text{where } (v_i, s_i) = \mathit{aux}(t_i) \text{ for } i = 1, \ldots, k.$$

$$\mathit{aux}(f(t_1, \ldots, t_k)) = \left(v, \{f(v_1, \ldots, v_k) \mapsto v\} \cup \bigcup_{i=1,\ldots,k} s_i\right)$$
$$\text{where } (v_i, s_i) = \mathit{aux}(t_i) \text{ for } i = 1, \ldots, k$$
$$\text{and } I \vdash f(v_1, \ldots, v_k) \mapsto v \text{ if such } v \text{ exists and } v = \mathit{default}_f \text{ otherwise.}$$

$$\mathit{aux}(v) = (v, \emptyset)$$

Fig. 6. $\mathit{flatten}_I(A)$ flattens function entries to be inserted into I given a nested ground atom A. If the output type of f is N, then $\mathit{default}_f$ is a fresh constant from N; otherwise it is $\bot$. The auxillary function *aux* takes a ground atom and returns the "output value" of the ground atom and the set of flattened facts it will populate.

and let $A[\sigma]$ denote the ground atom obtained by applying σ to atom A in the standard way. Given an egglog program P consisting of a set of rules and $I = (DB, \equiv)$, then $T_P^{\uparrow}(I) = DB \cup T_P(I)$ and $T_P(I) = (DB', \equiv)$, where

$$DB' = \bigcup_{(A \text{ :- } A_1, \ldots, A_m) \in P} \left\{\mathit{flatten}_I(A[\sigma]) \mid \forall_{i=1,\ldots,m}\, I \vdash A_i[\sigma]\right\}$$

However, functions in $T_P^{\uparrow}(I)$ may no longer preserve the functional dependencies, as it is possible that the same key $(v_1, \ldots, v_k)$ are mapped to more than one v in some f. We call $T_P^{\uparrow}(I)$ a *pre-instance*, since it is not a valid instance yet. To transform a pre-instance into a valid instance, we further define the rebuilding operator $R((DB, \equiv)) = (DB_R, \equiv_R)$, where:

$$(\equiv_R) = \text{equivalence closure of } \left((\equiv) \cup \left\{(n_1, n_2) \;\middle|\; \begin{array}{l} f(v_1, \ldots, v_k) \mapsto n_1 \in DB, \\ f(v_1, \ldots, v_k) \mapsto n_2 \in DB, \\ n_1, n_2 \in N \end{array}\right\}\right)$$

$$DB_R = \lambda_{\equiv_R}\left(\left\{f(v_1, \ldots, v_k) \mapsto \mathit{merge}_{f,\equiv}(K) \;\middle|\; \begin{array}{r} K = \{v : f(v_1, \ldots, v_k) \mapsto v \in DB\} \\ \text{and } K \text{ is not empty} \end{array}\right\}\right)$$

$$\mathit{merge}_{f,\equiv}(K) = \begin{cases} \min(\lambda_{\equiv}(K)) & \text{if the output type of } f \text{ is } N; \\ \bigsqcup K & \text{if the output type of } f \text{ is } C. \end{cases}$$

Note that the canonicalization in computing DB_R (i.e., $\lambda_{\equiv_R}$) may cause I to be invalid again, so successive rounds of rebuilding may be needed. Therefore, the complete rebuilding function R^{∞} is defined as iterative applications of R until fixpoint. The rebuilding process always terminates, as it shrinks the size of the database in each iteration.

We define one iteration of evaluating an egglog program F_P as $R^{\infty} \circ T_P^{\uparrow}$, i.e., do rule application once, and apply rebuilding until fixpoint. Intuitively, applying F_P to a database makes it "larger", in the sense that more facts may be represented. To capture this monotonicity, we define the expanded

egglog to be compatible with existing egg applications, which can be non-monotonic. Instead, we define egglog semantics using the inflationary immediate consequence operator, which is used to describe semantics for non-monotonic extension of Datalog such as Datalog$^{\neg}$ [Kolaitis and Papadimitriou 1988].

database $E_{\equiv}(DB)$ such that $f(v_1, \ldots, v_k) \mapsto n \in E_{\equiv}(DB)$ iff $f(\lambda_{\equiv}(v_1), \ldots, \lambda_{\equiv}(v_k)) \mapsto \lambda_{\equiv}(n) \in DB$ and $f(v_1, \ldots, v_k) \mapsto c \in E_{\equiv}(DB)$ iff $\exists c'. f(\lambda_{\equiv}(v_1), \ldots, \lambda_{\equiv}(v_k)) \mapsto c' \in DB \wedge c \sqsubseteq c'$. A database is larger than or equal to another database if it knows at least as many facts and equalities, so we define $(DB_1, \equiv_1) \sqsubseteq_I (DB_2, \equiv_2)$ iff $E_{\equiv_1}(DB_1) \subseteq E_{\equiv_2}(DB_2)$ and $\equiv_1 \subseteq \equiv_2$.

Although F_P is not a monotone function in general, the following sequence of iterative applications is always monotonically increasing:

$$I_\perp \sqsubseteq_I F_P(I_\perp) \sqsubseteq_I F_P(F_P(I_\perp)) \sqsubseteq_I F_P(F_P(F_P(I_\perp))) \sqsubseteq_I \ldots$$

for initial database $I_\perp = (\emptyset, Id_{N \cup C})$ where $Id_{N \cup C}$ is the identity relation over $N \cup C$. This ensures the existence of a fixpoint.

Finally, the result of evaluating an egglog program P is defined as the inductive fixpoint of F_P, i.e., $[\![P]\!] = F_P^\infty(I_\perp)$. For many practical applications, $[\![P]\!]$ often has an infinite size, so we only calculate a finite under-approximation of the result, i.e., $(R^\infty \circ T_P^\uparrow)^n(I_\perp)$ for some iteration size n.

4.3 Semi-naïve Evaluation

Last section gives an algorithm for evaluating egglog programs, which iteratively apply the immediate consequence operator ($T_P^\uparrow$) and the rebuilding operator R^∞. We call this algorithm the naïve evaluation. Despite straightforward, the naïve evaluation may duplicate works by rediscovering same facts over and over again. The semi-naïve algorithm mitigates this problem by incrementalizing the evaluation. In semi-naïve evaluation, each iteration maintains a differential database ΔDB_i, which will only contain tuples that are updated or new in this iteration. The semi-naïve rule application operator $T_P^{SN}(I_i, \Delta DB_i)$ additionally takes a differential database. For each rule A :- $A_1, \ldots, A_m$, T_P^{SN} will expand it into m delta rules $\{A$:- $A_1, \ldots, A_{j-1}, \Delta A_j, A_{j+1}, \ldots, A_m \mid j \in 1 \ldots m\}$ and apply these rules to the database similar to T_P.

Algorithm 1 The semi-naïve evaluation of an egglog program.

procedure $F_P^{SN}(P, n)$
  $I_0 \leftarrow I_\perp$; $\Delta DB_0 \leftarrow \emptyset$;
  for $i = 1 \ldots n$ **do**
    $(DB_i, \equiv_i) \leftarrow R^\infty\left(I_{i-1} \cup T_P^{SN}\left(I_{i-1}, \Delta DB_{i-1}\right)\right)$;
    $\Delta DB_i \leftarrow DB_i - DB_{i-1}$;
    $I_i \leftarrow (DB_i, \equiv_i)$;
  return I_n;

We prove the following theorem in the full version [Zhang et al. 2023].

Theorem 4.1. *The semi-naïve evaluation of an* egglog *program produces the same database as the naïve evaluation.*

5 IMPLEMENTATION

egglog is implemented in approximately 4,200 lines of Rust [Rust [n.d.]]. The codebase is open-source.[5] egglog provides both a library interface and the text format shown in Section 3. As suggested by the previous sections, egglog's design and implementation takes many cues from modern Datalog implementations [Jordan et al. 2016; Sahebolamri et al. 2022; Szabó et al. 2018]. Below, we describe the design of egglog's core components, as well as some of the benefits of this design.

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

5.1 Components

egglog's main components are the database itself, rebuilding procedure, and the query engine.

Database. Unlike most other Datalog implementations, egglog is based on a functional database instead of a relational database. In other words, each function/relation is backed by a map instead of a set. This ensures that egglog can efficiently perform the "get-or-default" operation required to implement terms. For example, evaluating the term `(g x)` will first lookup `x` in the map for function `g`. If something is present, it will be returned, otherwise `g`'s **`:default`** expression is evaluated, placed in the map for `(g x)`, and returned.

The functional database also ensures that a function's inputs map to a single output. As discussed in Section 3.2, egglog uses a function's **`:merge`** expression to resolve conflicts in map. Function conflicts can arise in two ways: (1) the user or a rule expressly calls `(set (f x) y)` where `(f x)` is already defined, or (2) a **`union`** causes a function's inputs to become equivalent. The functional database allows for efficient detection of the first case; the second essentially requires computing congruence closure, which is done by the rebuilding procedure.

Rebuilding Procedure. egglog's rebuilding procedure is based on the rebuilding procedure from egg [Willsey et al. 2021], which is in turn based on the congruence closure algorithm from Downey et al. [1980]. The rebuilding procedure is responsible for canonicalizing the database, which resolves (and creates) the second form of function conflicts discussed above. Suppose that a function f maps two different inputs to two different outputs: so $f(a) \mapsto b$ and $f(c) \mapsto d$. Say that a and c have recently been **`union`**ed, and that a is canonical; rebuilding must update the entry $f(c) \mapsto d$, since it is no longer canonical. Canonicalizing $f(c) \mapsto d$ to $f(a) \mapsto d$ causes a conflict with the previously existing entry $f(a) \mapsto b$. To resolve conflicts, egglog uses the **`:merge`** expression to combine the two outputs into a single output. The **`:merge`** may end up **`union`**ing more things, which may in turn create more conflicts. The rebuilding procedure continues until no more conflicts are created. For functions where the **`:merge`** behavior is to **`union`** the two outputs, this is the same as congruence closure. For other **`:merge`** behavior, this is more akin to the e-class analysis propagation algorithm from Willsey et al. [2021].

Query Engine. Once the database is canonicalized, e-matching is reduced to a query over the database. Since egglog is based on Datalog, it can naturally use established techniques for efficient query execution. egglog's query engine is based on the relational e-matching technique from Zhang et al. [2022], which uses a worst-case optimal join algorithm called Generic Join [Ngo et al. 2018]. egglog also features some important optimizations on top of Zhang et al. [2022]'s implementation that are only possible because of egglog's database-native approach[6], such as the semi-naïve evaluation presented in Section 4.3

5.2 Language-based Design

Like most Datalog implementations (and *unlike* most EqSat implementations), egglog is designed primarily as a programming language. Users can write egglog programs in a text format (shown in Section 3), and the tool parses, typechecks, compiles, and executes them. The egglog language includes several base types (including 64-bit integers and strings) and operations over them. Users can also define their own types and operations by using the Rust library interface.

Compared to tools like egg that are more embedded in the host language, this design provides more of the user's program to the compiler for checking and optimization. For example, egg provides conditional rewrite rules that gate a rewrite on some condition. The guards are essentially

[6] Zhang et al. [2022]'s implementation still uses an e-graph data structure; it creates a database from the e-graph whenever it needs to e-match. egglog avoids this copying overhead since it is already a database.

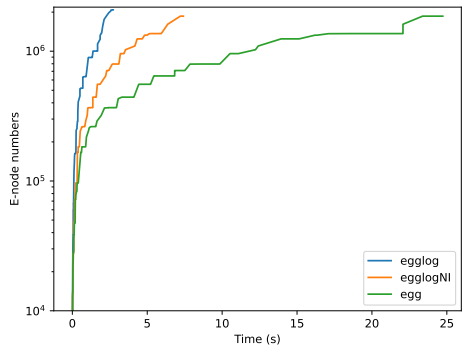


Fig. 7. Performance of egglog and egg on math benchmark. egglogNI grows the same e-graph with less time than egg. egglog explores a larger program space than both baselines thanks to semi-naïve evaluation.

Rust code, so egg cannot inspect them; it must just run the query and then check the guard. In egglog, there is no need for special conditional rewrites; all rules (and therefore rewrites) can have as many conditions in the query as needed. In addition, rewrite rules in egg are not typechecked; egglog prevents common errors by statically typechecking rules. The language-based approach also allows the user to better model their problem with as many datatypes, functions, and analysis as needed. egg is artificially limited to a single type and a single analysis in the e-graph due to its embedded nature; allowing for multiplicity would significantly complicate the generic types in egg's Rust implementation.

5.3 Micro-benchmarks

In this section, we evaluate the performance of egglog on a typical workload of equality saturation. Our two baselines are egg, a state-of-the-art equality saturation framework, and egglogNI, the non-incremental variant of egglog with semi-naïve evaluation disabled. We populate all three systems with a set of initial terms from egg's math test suite and grow the e-graph with rewrite rules using BackOff scheduler, the default scheduler of egg. We only use a subset of rules from math test suite that does not involve any analysis (so rules that require analyses like $x/x \longrightarrow 1$ if $x \neq 0$ are removed), because the scheduler behaves differently on analyses in the two systems[7]. As a result, egglogNI and egg produce the same e-graph in each iteration.

We ran each system for 100 iterations.[8] For each iteration, we ran three systems seven times and report the median time versus sizes of math e-nodes (tuples for egglog systems). The result is shown in Figure 7. Thanks to the efficient relational matching algorithm and the relational query optimizer, egglogNI grows the exact same e-graph in less time than egg, yielding a 3.34× speedup at the end of iteration 100. Moreover, egglog grows a slightly larger e-graph than egg with a 9.27× speedup, because its semi-naïve evaluation avoids redundant matches.

[7] egglog does not distinguish analyses rules from other rules, while egg treats e-class analyses specially and runs them to saturation in each iteration.

[8] All experiments in this paper are executed on a MacBook Pro with Apple M2 processor and 16GB memory.

6 CASE STUDIES

6.1 Unification-Based Points-to Analysis

Many program analysis tools [Balatsouras and Smaragdakis 2016; Bravenboer and Smaragdakis 2009; Whaley et al. 2005] are implemented in Datalog. The declarative nature of Datalog makes the development easier, and the mature relational query optimization and execution techniques provide competitive performance and sometimes lead to order-of-magnitude speedup [Bravenboer and Smaragdakis 2009].

Points-to analysis computes an over-approximation of the set of possible allocations a pointer can point to at run time. Most points-to analyses implemented in Datalog are subset based (i.e., Andersen style). These analyses are precise, but they scale poorly due to its quadratic complexity. On the other hand, unification-based points-to analysis (i.e., Steensgaard style [Steensgaard 1996]) is nearly linear in complexity and therefore scales much better, at the cost of potential imprecision. In a unification-based analysis, if it is ever learned that p may point to two allocations a_1 and a_2, the allocations are *unified* and considered equivalent. The points-to relation in a Steensgaard analysis is essentially a function from pointers to the equivalence class of allocations they point to. This is less precise than subset-based analysis, but it is more scalable, because it avoids tracking every individual allocation a pointer points to.

However, despite its success in hosting other program analysis algorithms, classical Datalog fails to express Steensgaard analysis efficiently due to the lack of support for fast equivalence. A plain representation of the equivalence relation in Datalog is quadratic in space, which defeats the purpose of unifying points-to allocations of a pointer. Recently, Soufflé added support for union-find–backed relations to benefit from the space- and time-efficient representation in the union-find data structure [Nappa et al. 2019]. Relations marked with the `eqrel` keyword in Soufflé will be stored using union-find, so the equivalence closure property of the relation will be automatically maintained, without explicit rules like transitivity, which is quadratic. However, `eqrel` relations in Soufflé only *maintain* equivalence relations efficiently, but fail to interact with the rest of the rules efficiently. Therefore, practical Steensgaard analyses do not use the equivalence relation directly. Consider this rule[9] adapted from the Steensgaard analysis benchmark in the `eqrel` paper [Nappa et al. 2019]:

```
// *x = y; p = *q; x and q points to the same set of allocs
eql(yAlloc, pAlloc) :- store(x, y), vpt(x, xAlloc), vpt(y, yAlloc),
                       load(p, q), vpt(p, pAlloc), vpt(q, qAlloc),
                       eql(xAlloc, qAlloc),
```

where `vpt` is the points-to relation from pointers to allocations, and `eql` is the equivalence relation declared with `eqrel`. To see the performance of this rule, let us consider the subquery `vpt(x, xAlloc), vpt(q, qAlloc), eql(xAlloc, qAlloc)`. To evaluate this subquery, Soufflé has to join over the `eql` relation, even when it is known that `xAlloc` and `qAlloc` are equivalent. We call this additional join over the equivalence relation "join modulo equivalence". This occurs frequently when using equivalence relations in practice and often leads to unacceptable performance. `egglog` eliminates this join modulo equivalence by actively canonicalizing each element to its canonical representation. Because two elements are equivalent if and only if they have the same canonical representation, it suffices for `egglog` to perform only an equality join over `vpt(x, xAlloc)` and `vpt(q, qAlloc)` with `qAlloc = xAlloc`, without joining with an auxillary quadratic relation.

`cclyzer++` [Barrett and Moore 2013] implements Steensgaard analyses in Datalog with extensions, Soufflé in particular. Joins like the above are too expensive for `cclyzer++`, so `cclyzer++`

[9] In contrast to our paper, Nappa et al. [2019] views `vpt` itself as an `eqrel` relation, and the body of the rule joins over only `store`, `vpt`, `load`. Our presentation here is adjusted to be consistent with `cclyzer++`.

avoids such joins modulo equivalence as much as possible. In fact, profiling shows that the only rule that involves join modulo equivalence in `cclyzer++` is an order of magnitude slower than any other rule `cclyzer++` uses to compute the points-to analysis.

In Steensgaard analyses, all allocations pointed to by the same pointer should be unified, so only one allocation per pointer will need to be tracked, which ensures an almost linear performance. However, this key performance benefit is lost in a direct encoding of Steensgaard analyses in Datalog. For each pair of pointer `p` and the allocation it points to, a direct encoding will create a tuple `vpt(p, alloc)`, so `vpt` may contain many allocations pointed to by the same pointer, despite them all being equivalent. The many allocations pointed to by the same pointer will be further propagated to other pointers, causing a blow up in the points-to relation. To make sure only one representative per equivalence class will be propagated, `cclyzer++` uses a complex encoding with choice domain [Hu et al. 2021; Krishnamurthy and Naqvi 1988] and customize its own version of equivalence relation using subsumptive rules [Köstler et al. 1995].

Qualitatively, we argue such an encoding is complex and error-prone, and we identify two independent bugs related to the `cclyzer++` encoding. Each bug could lead to unsound points-to analysis result. To fix the bugs, we have to bring back the `eqrel` relations of Soufflé. In other words, our patched version involves the interaction among choice domain, subsumptive rules, and `eqrel` relations, three of the newest features of Soufflé. In our experience, the interplay of these features can produce unexpected results and is extremely tricky to debug.

Compared to the sophisticated and unintuitive encoding one has to develop to express efficient Steensgaard analyses in Datalog, writing Steensgaard analyses in `egglog` is straightforward. The user only needs to specify that the `vpt` relation is a function where functional dependency repair is done via unifying the violating ids, and `egglog` takes care of all the unification and canonicalization. Our insight here with `egglog` is that, if two terms are known to be equivalent, they should considered indistinguishable by the database. `egglog`'s canonicalization means we do not have to join modulo equivalence; a regular join suffices.

Benchmarking Points-To Analysis. We benchmark the performance of `egglog` on points-to analyses against three baselines:

- `eqrel` uses an explicit `eqrel` relation to represent the equivalences among allocations. In `eqrel`, because there is no canonical representation of pointers, a pointer may point to multiple (equivalent) allocations in `vpt`.
- `cclyzer++` uses the original encoding `cclyzer++` developed for Steensgaard analyses. It uses a custom equivalence relation that keeps record of the canonical representation of an allocation and avoids duplicated allocations with Soufflé's choice domain feature. However, `cclyzer++` has to perform joins modulo equivalence for analyzing the `load` instruction, and the custom equivalence relation is semantically unsound.
- `patched` is a patched encoding we developed based on `cclyzer++`'s encoding. We made the custom equivalence relation sound by bringing back the `eqrel` relation in Soufflé, while keeping the canonical representations of allocations. We also added an additional rule to address a congruence-related bug in `cclyzer++`.

Moreover, we compare against `egglogNI`, the non-incremental variant of `egglog` with semi-naïve disabled.

We reimplemented a subset of `cclyzer++` in `egglog` and three baselines. The points-to analyses we implement is context-, flow-, path-insensitive and field-sensitive. We ran points-to analyses written in two variants of `egglog` and the three baselines on programs from postgresql-9.5.2 with a timeout of 20 seconds. All the systems except for `cclyzer++` report the same size for computed points-to relations. Figure 8 shows the result. `eqrel` times out on all but one of the benchmarks,

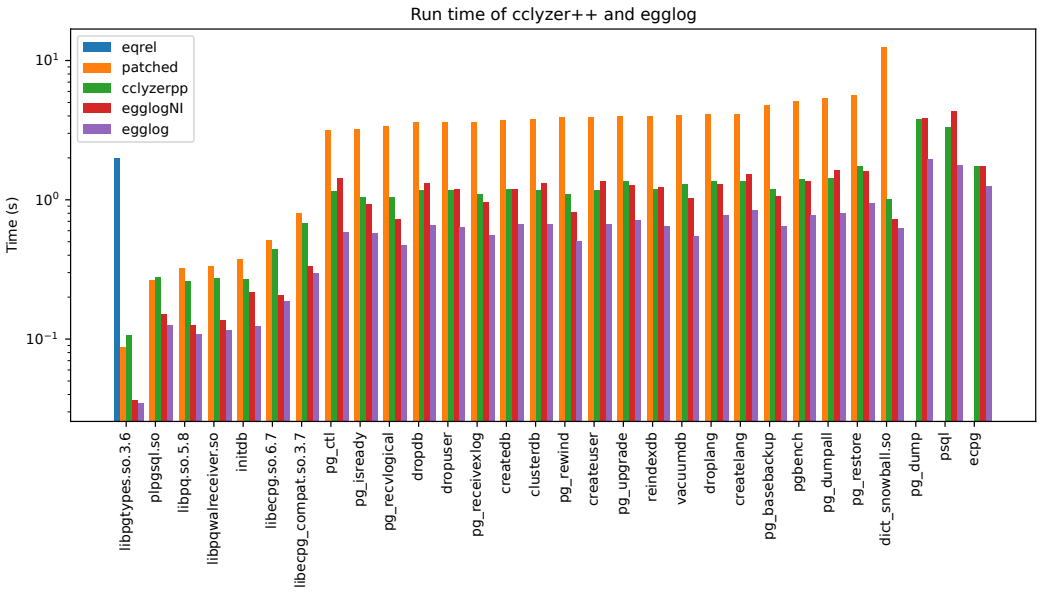


Fig. 8. Performance comparison between egglog and various encoding of Steensgaard analyses in Soufflé. Benchmarks that time out are not shown. In particular, eqrel times out on all but one benchmarks and cclyzer++ times out on the three benchmarks on the right.

and the patch to cclyzer++, despite making it sound, does make the encoding slower with the explicit equivalence relation and times out on three of the benchmarks. egglog is faster than all Soufflé based baselines. The comparison between egglog and egglogNI additionally shows that semi-naïve evaluation brings a substantial amount of speedup to the computation of points-to analyses by avoiding duplicated works. Not counting the timed-out benchmarks, egglog achieves a 4.96× speedup over patched on average, which is the fastest sound encoding in Soufflé available. Moreover, it achieves a 1.94× speedup over cclyzer++, and 1.59× over egglogNI.

6.2 Herbie: Making an EqSat Application Sound

Herbie [Panchekha et al. 2015] is a widely-used, open-source tool for making floating-point programs more accurate, with thousands of users and yearly stable releases. Herbie takes as input a real expression, and returns the most accurate floating-point implementation it can synthesize. Since floating-point error is a critical issue in scientific computing, Herbie is used in a variety of domains, including machine learning, computer graphics, and computational biology.

The core of Herbie's algorithm is to run equality saturation to explore equivalent programs. These programs are mathematically equivalent over the real numbers, but may have different behavior over floating-point numbers. Herbie considers candidate programs from the results of equality saturation, finding the most accurate among them.

Herbie's rewrite rules are known to be unsound, which has been the cause of numerous bugs in the past. In addition, unsound rules prevent Herbie from running equality saturation longer once unsoundness occurs. Unfortunately, merely removing the unsound rules makes Herbie useless on a large portion of its benchmark suite. For example, Figure 9b shows a rewrite rule which is critical to Herbie's ability to find more accurate programs. Using these unsound rules in a sound

$$\frac{a * b}{c} \implies \frac{a}{\frac{c}{b}}$$

(a) A fraction rule which requires $b \neq 0$.

$$x - y \implies \frac{x^3 - y^3}{x^2 + xy + y^2}$$

(b) A more complex rule derived from the factorization of $x^3 - y^3 = (x-y)(x^2+xy+y^2)$. This is sound if either $x \neq 0$ or $y \neq 0$.

Fig. 9. Herbie [Panchekha et al. 2015] uses rewrite rules to create program variants with less floating-point error from phenomena like cancellation. Some rules are only sound under certain conditions.

```
(function lo (Math) Rational :merge (max old new))
(function hi (Math) Rational :merge (min old new))

(rule ((= e (Sqrt a)))
      ((set (lo e) (rational 0 0))))

(rule ((= e (Sqrt a))
       (= loa (lo a)))
      ((set (lo e) (sqrt loa))))
```

Fig. 10. A few example rules for interval analysis of sqrt in egglog. The *lo* relation tracks the lower bound for each term, and we merge lower bounds by taking the max. Similarly, the *hi* relation tracks the upper bound. First, we know that root of anything is non-negative. Next, since taking the root is monotonic, we can propagate the bounds from the arguments directly to the bounds of the result.

way requires a more sophisticated analysis of Herbie's input programs. This analysis was nearly impossible with Herbie's existing e-graph implementation.

egglog has allowed us to implement precise analyses to make all rewrites sound. First, we implement an interval analysis in egglog, allowing rules to utilize information about the range of terms in the program (Figure 10). This unlocks a range of crucial rules involving division, including the rule shown in Figure 9a.

While the interval analysis enables many of Herbie's rules, it is not sufficient for some more difficult cases. We additionally implemented a "not equals to" analysis, which leverages both the interval analysis as well as facts inferred during rewriting. egglog's support for multiple interacting analyses enables cleanly separating interval and $\neq$ rules; in other EqSat frameworks they would be implemented as a fused, monolithic analysis requiring much significantly more complicated rules. The not-equals analysis allows Herbie to soundly solve one of its classic cancellation benchmarks: $\sqrt[3]{v+1} - \sqrt[3]{v}$. First, the interval analysis proves that $v + 1 \neq v$. Next, the rule $a \neq b \implies \sqrt{a} \neq \sqrt{b}$ implies $\sqrt[3]{v+1} \neq \sqrt[3]{v}$. This allows us to finally apply the rewrite from Figure 9b, substituting $\sqrt[3]{v+1}$ for x and $\sqrt[3]{v}$ for y, reducing the error of the expression from extremely high to near zero.

Figure 11 shows the results of our evaluation of Herbie using egglog. Herbie has a benchmark suite of 289 floating-point programs, collected from a variety of domains. We ran Herbie on each of these programs using both the unsound ruleset and egglog's sound analysis. Using egglog's analysis makes Herbie faster overall (73.91 minutes vs 81.91 minutes). This is because egglog generates no unsound programs, which slow down Herbie's search.

In 104 cases, Herbie using a sound analysis is actually able to find a more accurate program than Herbie using the unsound ruleset. In 135 cases, Herbie's unsound ruleset finds more accurate results than egglog's sound analysis. There are several outliers in Figure 11. The point on the far left represents a benchmark which Herbie using the unsound ruleset is unable to solve. This input program is $9x^4 - y^2(y^2 - 2)$, and the solution involves an algebraic rearrangement and fma (fused multiply-add) operation. The point on the far right represents a program that overflows egglog's rational type, which can be easily fixed in the future.

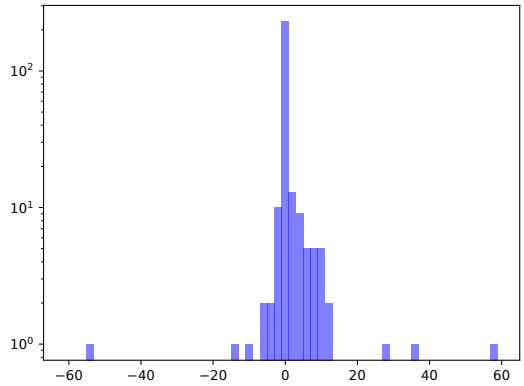


Fig. 11. Graph showing the difference in error between Herbie using egglog's sound analysis and Herbie using the unsound ruleset across all of Herbie's benchmark suite. The horizontal axis is the difference in the average bits of error using Herbie's unsound rules vs. egglog's sound rules. The vertical axis is the number of benchmarks. Negative values represent benchmarks in which Herbie found a more accurate program using egglog's analysis.

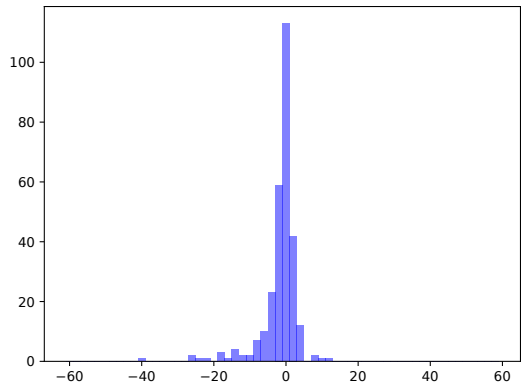


Fig. 12. Graph showing the difference in runtime (in seconds) between Herbie using egglog's sound analysis and Herbie using the unsound ruleset across all of Herbie's benchmark suite. The horizontal axis is the difference in time to execute the benchmark. The vertical axis is the number of benchmarks. Negative values represent benchmarks in which Herbie was faster using egglog's analysis.

7 RELATED WORK

E-graphs and Equality Saturation. E-graphs were first introduced by Nelson [1980] in late 1970s to support a decision procedure for the theory of equalities. Downey et al. [1980] later introduced a more efficient algorithm and analyzed the its time complexity. E-graphs are used at the core of many theorem provers and solvers [Barrett et al. 2011; De Moura and Bjørner 2008; Detlefs et al. 2005]. Because e-graphs can compactly represent program spaces, they were repurposed for program optimization in the 2000s [Joshi et al. 2002; Tate et al. 2009]. Other data structures for compact program space representations are developed in parallel, including finite tree automata [Wang et al. 2017a,b] and version space algebras [Polozov and Gulwani 2015; Wolfman et al. 2001]. There are two essential problems to these data structures: how to construct the desired program space and

how to search it. Tate et al. [2009] observed that the program space can be grown via equational, non-destructive rewrites, which they called equality saturation. This insight leads to a line of work on using equality saturation for program optimization and program synthesis [Nandi et al. 2020, 2021; Panchekha et al. 2015; VanHattum et al. 2021; Wang et al. 2020; Yang et al. 2021]. However, a problem with this rewriting-based approach to program space construction is that, in many cases, sound rewrite rules are difficult to define in a purely syntactic way. The egg framework by Willsey et al. [2021] mitigates this issue by introducing e-class analyses, which allow simple semantic analyses over the e-graphs. Our work improves e-class analyses by allowing more expressive analysis rules to be defined compositionally.

Zhang et al. [2022] first studied the connection between e-graphs and relational databases. By reducing pattern matching over e-graphs to relational queries, they made the matching procedure orders of magnitude faster. However, their technique has the dual representation problem, i.e., one has to keep both the e-graph and its relational representation, which limits its practical adoptions. We build on their work and view the entire equality saturation algorithm from the relational perspective. This saves us from synchronizing two representations of e-graphs and further exploits the performance benefits of the relational approach.

`egglog` also brings new insights on some problems in e-graphs and equality saturation. For example, the literature studied the problem of incremental pattern matching over e-graphs. Zhang et al. [2022] conjectured that this problem can be solved by classical techniques of incremental view maintenance in databases. We complement their argument with a concrete implementation of incremental matching using semi-naïve evaluation [Balbin and Ramamohanarao 1987]. Moreover, the literature studied the "proof" problem on e-graphs [Flatt et al. 2022a; Nieuwenhuis and Oliveras 2005]: many domains require not only the optimized terms that are equivalent to the original terms, but also a proof why they are equivalent. A future direction is to study proof generations for `egglog` programs.

Datalog and Relational Databases. Functional dependency repairs via lattice joins in `egglog` is directly inspired by Flix [Madsen et al. 2016]. Flix extends Datalog by allowing relations to be optionally annotated by lattices. With this feature, Flix is able to express many advanced program analyses algorithms efficiently. `egglog` can simulate Flix programs by setting `:merge` to the lattice join operator. Flix can be regarded as among the works that try to find a theoretical foundation for recursive aggregates [Abo Khamis et al. 2022; Köstler et al. 1995; Ross and Sagiv 1992; Van Gelder 1992]. A similar lattice-based approach to recursive aggregates is studied by Bloom^L [Conway et al. 2012]. Other Datalog systems that support recursive aggregates include LogicBlox [Aref et al. 2015] and Rel [developers [n.d.]].

Rewrite rules in `egglog` generalize rules in Datalog because the heads of rules can generate fresh ids. This is called tuple-generating dependencies (TGDs) in the database literature. Moreover, while functional dependencies has the form $R(x_1, \ldots, x_i, \ldots, x_k), R(x_1, \ldots, x_i', \ldots x_k) \rightarrow x_i = x_i'$ equality-generating dependencies (EGDs) generalize functional dependencies by allowing equalities between different columns in different relations. A family of algorithms called the chase can be used to reason about both TGDs and EGDs [Benedikt et al. 2017; Deutsch et al. 2008]. Moreover, the model semantics of the chase directly gives a model semantics of a subset of `egglog` where union is the only `:merge` operation. Compared to general TGDs and EGDs, `egglog` imposes syntactic constraints over the programs so that rules applications are deterministic and efficient. Datalog$^\pm$ [Calì et al. 2009] is a family of extensions to Datalog based on TGDs and EGDs for ontological reasoning.

Concurrent to our work and independent to the work on the chase, Bidlingmaier [2023a] formalizes Datalog with equality, which shares the same core idea to `egglog`, as relational Horn logic and partial Horn logic and studies its properties from a categorical point of view. Bidlingmaier [2023b]

further describes an evaluation algorithm for Datalog with equality similar to the chase. Different from theirs, our work is motivated by practical applications in program optimization and program analysis, and we focus on a simpler operational model of `egglog`.

While termination for Datalog with a variety of extensions is well studied, the termination condition of `egglog` is quite open. In the future, we hope to better understand the termination of `egglog` by further studying the connection between `egglog` and the chase. For example, the database theory community has established many conditions for chase termination (e.g., Bellomarini et al. [2018]; Calì et al. [2009]; Fagin et al. [2003]), and one could potentially apply these results to `egglog` by translating `egglog` rewrite rules and functional dependencies into TGDs and EGDs. On the other hand, nearly all instantiations of equality saturation in practice will diverge . Being a generalization of equality saturation, `egglog` allows for divergence by design.

A key feature of `egglog` is its efficient equational reasoning. Although equational reasoning can be expressed in Datalog with an explicit equivalence relation, doing so is very inefficient. The `patched` baseline in Section 6.1 is a slightly more efficient encoding of equational reasoning in Datalog using Soufflé extensions including choice domain and subsumptive rules. We also attempted several other approaches to optimize equational reasoning with existing features such as recursive aggregates and Constraint Handling Rule's simpagation rules [Frühwirth 1998]. However, we found none of these encodings provide a natural abstraction nor competent performance.

Logic Programming and Automated Theorem Proving. `egglog` bears some similarity with logic programming languages like Prolog. Similar to unification variables in Prolog, fresh ids in `egglog` can represent unknown information (see, e.g., Zhang et al. [2023, Appendix A.1]), and the congruence closure can be viewed as a dual procedure to unification [Kanellakis and Revesz 1989]. In Zhang et al. [2023, Appendix A.3], we also show an implementation of a Hindley-Milner type inference algorithm, of which the key construct is the unification mechanism implemented as a few `egglog` rules.

However, several distinguishing features make `egglog` highly efficient for its target domains, namely program analysis and optimization. For example, `egglog` does not allow backtracking, so its union-find data structure does not need to be backtrackable or persistent (unlike in Prolog or SMT solvers), which makes it efficient for tasks that are monotone in nature (e.g., equality saturation and pointer analysis). Moreover, `egglog` uses a bottom-up evaluation algorithm more similar to Datalog than Prolog (top-down backtracking search). One way of (partially) viewing `egglog` is as a logic programming language that combines the bottom-up evaluation of Datalog and the unification mechanism of Prolog. The "magic-set transformation" is a closely related technique to simulate top-down evaluation in a bottom-up language in a demand-driven fashion. We show in the full version [Zhang et al. 2023] several pearls that uses this idea to simulate top-down evaluations. On the other hand, Prolog has imperative features such as `cut`, which removes choice points. `egglog` does not have a direct analog of `cut` (because `egglog` does not backtrack), although `egglog` has other imperative features borrowed from EqSat techniques that makes fine control of the execution such as rule scheduling.

SMT solvers are powerful tools for deciding combinations of logic theories and automatically proving theorems [Barrett et al. 2011; De Moura and Bjørner 2008]. Many rewrite rules in `egglog` can be expressed as SMT axioms. In fact, SMT solvers support a richer language than `egglog` with features like disjunction and built-in theories like the theory of integer programming. However, a key difference between `egglog` and SMT solvers is that the output of `egglog` is *minimal* (or universal in database terminology). While it is possible to "hack into" an SMT solver to repurpose it as an EqSat engine [Flatt et al. 2022b], such techniques are arcane and not officially supported. EqSat and `egglog`'s native support for extraction makes them better suited for program optimization.

Recently, researchers have extended Datalog to dispatch more complex constraints to be solved by an SMT solver [Bembenek et al. 2020]. This greatly extends the reach of Datalog, allowing the user to specify constraints in a variety of theories and logics. In `egglog` we emphasize efficiency, and chose more conservative extensions that can be implemented by fast data structures like union-find.

8 CONCLUSION

`egglog` unifies both Datalog and EqSat style fixpoint reasoning. From the perspective of a Datalog programmer, `egglog` adds fast and extensible equivalence relations that still support key database optimizations like query planning and semi-naive evaluation. From the perspective of an EqSat user, `egglog` adds composable analyses, extensible uninterpreted functions, and incremental e-matching, thus significantly simplifying complex conditional rewrites and scalable program analyses. `egglog`'s novel *merge expressions* for user-specified functional dependency repair are the key technical mechanism enabling this synthesis of fixpoint reasoning capabilities.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We are grateful to Martin Bidlingmaier for sharing his insights on `EqLog`, a concurrent work to `egglog`, to Martin Bravenboer for discussions on the `Rel` programming language, to Scott Moore and Langston Barret for answering questions about `cclyzer++`, and to friends at the UW PLSE group for their feedback on the early draft. This material is based upon work supported by the National Science Foundation under Grant No. 1749570, by the U.S. Department of Energy under Award Number DE-SC0022081, by DARPA under contract FA8650-20-2-7008, and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. 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

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

George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

Isaac Balbin and Kotagiri 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 Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177.

Langston Barrett and Scott Moore. 2013. cclyzer++. https://github.com/GaloisInc/cclyzerpp.

Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-Based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.* 11, 9 (may 2018), 975–987. https://doi.org/10.14778/3213880.3213888

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 141:1–141:31. https://doi.org/10.1145/3428209

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

Martin E. Bidlingmaier. 2023a. Algebraic Semantics of Datalog with Equality. arXiv:2302.03167 [cs.LO]

Martin E. Bidlingmaier. 2023b. An Evaluation Algorithm for Datalog with Equality. arXiv:2302.05792 [cs.PL]

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. In *Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Providence, Rhode Island, USA) *(PODS '09)*. Association for Computing Machinery, New York, NY, USA, 77–86. https://doi.org/10.1145/1559795.1559809

Alessandro Cheli. 2021. Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation. *Journal of Open Source Software* 6, 59 (2021), 3078. https://doi.org/10.21105/joss.03078

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 *Automated 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 *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

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

Rel developers. [n.d.]. Rel reference. https://docs.relational.ai/rel/ref/overview

Soufflé Developers. [n.d.]. Soufflé Algebraic Data Types. https://souffle-lang.github.io/types#algebraic-data-types-adt. Accessed: 2022-11-01.

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (1 Oct. 1980), 758–771. https://doi.org/10.1145/322217.322228

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2003. Data Exchange: Semantics and Query Answering. In *Database Theory — ICDT 2003*, Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–224.

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022a. Small Proofs from Congruence Closure. In *Proceedings of The 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD '22)*, Vol. 3. TU Wien Academic Press, 75. https://doi.org/10.34727/2021/isbn.978-3-85448-053-2_13

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022b. Small Proofs from Congruence Closure. *CoRR* abs/2209.03398 (2022). https://doi.org/10.48550/arXiv.2209.03398 arXiv:2209.03398

Thom Frühwirth. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming* 37, 1 (1998), 95–138. https://doi.org/10.1016/S0743-1066(98)10005-5

Xiaowen Hu, Joshua Karp, David Zhao, Abdul Zreika, Xi Wu, and Bernhard Scholz. 2021. The Choice Construct in the Soufflé Language. In *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 163–181. https://doi.org/10.1007/978-3-030-89051-3_10

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. *SIGPLAN Not.* 37, 5 (May 2002), 304–314. https://doi.org/10.1145/543552.512566

Paris C. Kanellakis and Peter Z. Revesz. 1989. On the relationship of congruence closureand unification. *Journal of Symbolic Computation* 7, 3 (1989), 427–444. https://doi.org/10.1016/S0747-7171(89)80018-5 Unification: Part 1.

Thomas Koehler, Phil Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. *arXiv preprint arXiv:2111.13040* (2021).

Phokion G. Kolaitis and Christos H. Papadimitriou. 1988. Why Not Negation by Fixpoint?. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Texas, USA) *(PODS '88)*. Association for

Computing Machinery, New York, NY, USA, 231–239. https://doi.org/10.1145/308386.308446

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

Ravi Krishnamurthy and Shamim A. Naqvi. 1988. Non-Deterministic Choice in Datalog. In *JCDKB*.

Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. *SIGPLAN Not.* 51, 6 (jun 2016), 194–208. https://doi.org/10.1145/2980983.2908096

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. 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, 31–44. https://doi.org/10.1145/3385412.3386012

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

Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Fast Parallel Equivalence Relations in a Datalog Compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 82–96. https://doi.org/10.1109/PACT.2019.00015

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

Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.

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. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

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

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

Rust. [n.d.]. Rust programming language. https://www.rust-lang.org/. https://www.rust-lang.org/

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*. 77–88.

Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2022. Optimizing Tensor Programs on Flexible Storage. https://doi.org/10.48550/ARXIV.2210.06267

Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded* (Oxford, UK) *(Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 32–41. https://doi.org/10.1145/237721.237727

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (oct 2018), 29 pages. https://doi.org/10.1145/3276509

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)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Allen Van Gelder. 1992. The Well-Founded Semantics of Aggregation. 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, 127–138. https://doi.org/10.1145/137097.137854

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

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. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 62 (oct 2017), 26 pages. https://doi.org/10.1145/3133886

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. *Proceedings of the VLDB Endowment* (2020).

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 97–118. https://doi.org/10.1007/11575467_8

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

Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems.* arXiv:2101.01332

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. https://github.com/pldi23-eqlog-ae/paper/blob/main/paper.pdf

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

```
.type Tree = Leaf {} | Node {t1: Tree, t2: Tree}
.decl tree_size_demand(l: Tree)
.decl tree_size(t: Tree, res: number)
// populate demands from roots to leaves
tree_size_demand(t1) :-
    tree_size_demand($Node(t1, t2)).
tree_size_demand(t2) :-
    tree_size_demand($Node(t1, t2)).
// calculate bottom-up
tree_size($Node(t1, t2), s1 + s2) :-
    tree_size_demand($Node(t1, t2)),
    tree_size(t1, s1),
    tree_size(t2, s2).
tree_size($Leaf(), 1).
// compute size for a particular tree
tree_size_demand($Node($Leaf(), $Leaf())).
```

(a) tree_size in Soufflé

```
(datatype Tree (Leaf) (Node Tree Tree))
(datatype Expr (Add Expr Expr) (Num i64))
(function tree_size (Tree) Expr)
;; compute tree size symbolically
(rewrite (tree_size (Node t1 t2))
    (Add (tree_size t1) (tree_size t2)))
;; evaluate the symbolic expression
(rewrite (Add (Num n) (Num m))
    (Num (+ n m)))
(union (tree_size (Leaf)) (Num 1))
;; compute size for a particular tree
(define two (tree_size (Node (Leaf) (Leaf))))
```

(b) tree_size in egglog

Fig. 13. Computing tree size with Soufflé and egglog

A EGGLOG BY EXAMPLE

In this section, we will walk through a list of examples showing the wide applications of egglog.

A.1 Functional Programming with egglog

egglog is capable of evaluating many functional programs very naturally. The standard evaluation of Datalog programs is traditionally done bottom up. Starting from the facts known, it iteratively derives new facts, and when it terminates, all the derivable facts are contained in the database. In the evaluation of functional programs, however, we usually start with the goal of evaluating a big expression and break it down into smaller goals while traversing the abstract syntax tree from root to leaves, before collecting the results bottom up. In order to simulate the top-down style of evaluation of functional programs, Datalog programmers need to create manual "demand" relations that carefully tune the firing of rules to capture the evaluation order of functional programs. On the other hand, egglog express many functional programs very naturally, thanks to the unification mechanism.

For example, consider the task of computing the relation tree_size, which maps trees to their sizes. A full instantiation of the tree_size finds the size of *all* trees and therefore is infinite, so bottom-up evaluations will not terminate in languages like Soufflé. We need to manually *demand transform* the program to make sure we only instantiate tree_size for the trees asked for and their children. Demand transformation first populates a "demand" relation, and tree_size will compute only trees that resides in the demand relation. The program is shown in Figure 13a. To get the size of a specific tree, we have to first insert the tree object into the tree_size_demand relation to make a demand, before looking up tree_size for the actual tree size.

Similar to Datalog, egglog programs are evaluated bottom up. However, we do not need a separate demand relation in egglog, because we can use ids to represent unknown or symbolic information. To query the size of a tree t, we simply put the atom (tree_size t) in the action. egglog will create a fresh id as a placeholder for the value tree_size maps to on t, and the rest of the rules will figure out the actual size subsequently. The egglog program is shown in Figure 13b.

Conceptually, we create a "hole" for the value `(tree_size t)` is mapped to. A series of rewriting will utlimately fill in this hole with concrete numbers. We use fresh ids here in a way that is similar to how logic programming languages use logic variables. In logic programming, logic variables represent unknown information that has yet to be resolved. We view this ability to represent the unknown as one of the key insights to both `egglog` and logic programming languages like Prolog and miniKanren. However, unlike Prolog and miniKanren, `egglog` does not allow backtracking in favor of monotonicity and efficient evaluations.

A.2 Simply Typed Lambda Calculus

Previous equality saturation applications use lambda calculus as their intermediate representation for program rewriting [Koehler et al. 2021; Schleich et al. 2022; Willsey et al. 2021]. To manipulate lambda expressions in e-graphs, a key challenge is performing capture-avoiding substitutions, which requires tracking the set of free variables. A traditional equality saturation framework will represent the set of free variables as an e-class analysis and uses a user-defined applier to perform the capture-avoiding substitution, both written in the host language (e.g., Rust). As a result, users need to reason about both rewrite rules and custom Rust code to reason about their applications.

We follow the `lambda` test suite of `egg` [Willsey et al. 2021] and replicate the lambda calculus example in `egglog`. Instead of writing custom Rust code for analyses, we track the set of free variables using standard `egglog` rules. Figure 14 defines a function that maps terms to set of variable names. Since the set of free variables can shrink in equality saturation (e.g., rewriting $x - x$ to 0 can shrink the set of free variables from $\{x\}$ to the empty set), we define the merge expression as set intersections. The first two rules say that values have no free variables and variables have themselves as the only free variable. The free variables of lambda abstractions, let-bindings, and function applications are inductively defined by constructing the appropriate variable sets at the right hand side. Finally, the last three rewrite rules perform the capture-avoiding substitution over the original terms depending on the set of free variables. When the variable of lambda abstraction is contained in the set of free variables of the substituting term, a new fresh variable name is needed. We skolemize the rewrite rule so that the new variable name is generated deterministically. Note that the last two rules depend both positively and negatively on whether the set of free variables contains a certain variable, so this program is not monotonic in general.

`egglog` can not only express e-class analyses, which are typically written in a host language like Rust, but also semantic analyses not easily expressible in e-class analyses. For example, consider an equality saturation application that optimizes matrix computation graphs and uses lambda calculus as the intermediate representation. Users may want to extract terms with the least cost as the outputs of optimizations, but a precise cost estimator may depend on the type and shape information of an expression (e.g., the dimensions of matrices being multiplied). Expressing type inference within the abstraction of e-class analyses is difficult: in e-class analyses, the analysis values are propagated bottom up, from children to parent e-classes. However, in simply typed lambda calculus, the same term may have different types depending on the typing context, so it is impossible to know the type of a term without first knowing the typing context. Because the typing contexts need to be propagated top down first, e-class analysis is not the right abstraction for type inference. In contrast, we can do type inference in `egglog` by simply encoding the typing rule for simply typed lambda calculus in a Datalog style: we first break down larger type inference goals into smaller ones, propagate demand together with the typing context top down, and assemble parent terms' types based on the children terms' bottom up.

Figure 15 shows a subset of rules that perform type inference over simply typed lambda calculus. We determine the types of variables based on contexts. For lambda expressions, we rewrite the type of `(Lam x t1 e)` to be $t_1 \rightarrow t_2$, where t_2 is the type of e in the new context where x has

```
(function free (Term) (Set Ident)
    :merge (set-intersect old new))

;; Computing the set of free variables
(rule ((= e (Val v)))
      ((set (free e) (empty))))
(rule ((= e (Var v)))
      ((set (free e) (set-singleton v))))
(rule ((= e (Lam var body))
       (= (free body) fv))
      ((set (free e) (set-remove fv var))))
(rule ((= e (Let var e1 e2))
       (= (free e1) fv1) (= (free e2) fv2))
      ((set (free e) (set-union fv2
          (set-remove fv1 var)))))
(rule ((= e (App e1 e2))
       (= (free e1) fv1) (= (free e2) fv2))
      ((set (free e) (set-union fv1 fv2))))

;; [e2/v1]λv1.e1 rewrites to λv1.e1
(rewrite (subst v e2 (Lam v e1))
         (Lam v body))
;; [e2/v2]λv1.e1 rewrites to λv1.[e/v2]e1
;; if v1 is not in free(e2)
(rewrite (subst v2 e2 (Lam v1 e1))
         (Lam v1 (subst v2 e2 e1))
    :when ((!= v1 v2)
           (set-not-contains (free e2) v1)))
;; [e2/v2]λv1.e1 rewrites to λv3.[e/v2][v3/v1]e1
;; for fresh v3 if v1 is in free(e2)
(rule ((= expr (subst v2 e2 (Lam v1 e1)))
       (!= v1 v2)
       (set-contains (free e2) v1))
      ((define v3 (Skolemize expr))
       (union expr (Lam v3 (subst v2 e2
           (subst v1 (Var v3) e1))))))
```

Fig. 14. Free variable analysis and capture avoiding substitution in egglog. We use skolemization function `Skolemize` to deterministically generate fresh variables for capture-avoiding substitution.

```
(function typeof (Ctx Expr) Type)

(function lookup (Ctx Ident) Type)
(rewrite (lookup (Cons x t ctx) x) t)
(rewrite (lookup (Cons y t ctx) x)
         (lookup ctx x)
    :when ((!= x y)))

;; Type of matrix constants
(rewrite (typeof ctx (fill (Num n) (Num m) val))
         (TMat n m))

;; Type of variables
(rewrite (typeof ctx (Var x) )
         (lookup ctx x))

;; Type of lambda abstractions
(rewrite (typeof ctx (Lam x t1 e))
         (Arr t1 (typeof (Cons x t1 ctx) e)))

;; Populate type inference demand for
;; subexpressions of function application
(rule ((= (typeof ctx (App f e)) t2))
      ((typeof ctx f)
       (typeof ctx e)))

;; Type of function application
(rule ((= (typeof ctx (App f e)) t)
       (= (typeof ctx f) (Arr t1 t2))
       (= (typeof ctx e) t1))
      ((union t t2)))
```

Fig. 15. Type inference for simply typed lambda calculus with matrices. Here we require that lambda abstractions are annotated with parameter types. Section A.3 looses this restriction.

type t_1 (i.e., `(typeof (Cons x t1 ctx) e)`). Finally, because we cannot directly rewrite the type of function applications in terms of types of their subexpressions, we explicitly populate demands for subexpressions and derive the types of function applications using the types of subexpressions once they are computed.

A.3 Type Inference beyond Simply Typed Lambda Calculus

egglog is suitable for expressing a wide range of unification-based algorithms including equality saturation (Section 3.4) and Steensgard analyses (Section 6.1). In this section, we show an additional

example on the expressive power of egglog: type inference for Hindley-Milner type systems. Unlike the simple type system presented in Section A.2, a Hindley-Milner type system does not require type annotations for variables in lambda abstractions and allows let-bound terms to have a *scheme* of types. For example, the term `let f = \x. x in (f 1, f True)` is not typeable in simply typed lambda calculus, since this requires `f` to have both type $Int \to Int$ and $Bool \to Bool$. In contrast, A Hindley-Milner type system will accept this term, because both $Int \to Int$ and $Bool \to Bool$ are instantiations of the type scheme $\forall \alpha, \alpha \to \alpha$.

Concretely, to infer a type for the above term, a type inference algorithm will first introduce a fresh type variable α for `x`, the argument to function `f`, and infer that the type of `f` is $\alpha \to \alpha$. Next, because `f` is bound in a let expression, the algorithm generalizes the type of `f` to be a scheme by introducing forall quantified variables, i.e., $\forall \alpha . \alpha \to \alpha$. At the call site of `f`, the type scheme is instantiated by consistently substituting forall quantified type variables with fresh ones, and the fresh type variables are later unified with concrete types. For example, `f` in function application `f 1` may be instantiated with type $\alpha_1 \to \alpha_1$. Because integer `1` has type *Int*, type variable α_1 is unified with *Int*, making the occurrence of `f` here have type $Int \to Int$. The final type of `f 1` is therefore *Int*.

The key enabler of Hindley-Milner inference is the ability to unify two types. To do this, an imperative implementation like Algorithm W [Milner 1978] needs to track the alias graphs among type variables and potentially mutating a type variable to link to a concrete type, which requires careful handling. In contrast, egglog has the right abstractions for Hindley-Milner inference with the built-in power of unification. The unification mechanism can be expressed as a single injectivity rule

```
(rule ((= (Arr fr1 to1) (Arr fr2 to2)))
      ((union fr1 fr2)
       (union to1 to2)))
```

This rule propagates unification down from inductively defined types to their children types. At unification sites, it suffices to call `union` on the types being unified. For instance, calling `union` on `(Arr (TVar x) (Int))` and `(Arr (Bool) (TVar y))` will unify type variable x (resp. y) and *Int* (resp. *Bool*) by putting them into the same e-class.

Figure 16 shows a snippet of Hindley-Milner inference in egglog. We translate the typing rule to rewrite rules in egglog straightforwardly. The egglog rule for lambda abstractions says, whenever we see a demand to check the type of `\x.e`. We create a fresh type scheme `fresh-tv` with no variables quantified, binding it to `x` in the context, and infer the type of body as `t1`. Finally, we unify the type of `\x.e` with `fresh-tv → t1`. For function application `f e`, we can compact the two rules for function application in simply typed lambda calculus into one rules thanks to the injectivity rule: we simply equate the type `t1` of `f` and the arrow type `Arr t t2` for type `t` of `f e` and type `t2` of `e`, and injectivity will handle the rest of unifications. Finally, the rule for type inferring `let x = e1 in e2` will first get and, generalize[10] the type of `e1` in the current context, bind variable `x` to it and infer the type of `e2` as `t2`. The type of `let x = e1 in e2` is then unified with the type of `t2`.

In Hindley-Milner type systems, a type variable may be accidentally unified with a type that contains it, which results in infinitary types like $\alpha \to \alpha \to \ldots$ and is usually not what users intend to do. A Hindley-Milner inference algorithm will also do an "occurs check" before unifying a type variable with a type. In egglog, the occurs check can be done modularly, completely independent of the unification mechanism and the type inference algorithm. In Figure 16, we define an `occurs-check` relation and match on cases where a type variable is unified with an inductive type like the arrow

[10]Generalization, as well as instantiation for the rule for type inferring variables is a standard operation in type inference literature. They convert between types and type schemes based on contexts. We omit them from the presentation for brevity. To implement them, we also track the free type variables for each type in our implementation.

```
(function generalize (Ctx Type) Scheme)
(function instantiate (Scheme) Type)
(function lookup (Ctx Ident) Scheme)
(function typeof (Ctx Expr i64) Type)

;; Injectivity of unification
(rule ((= (Arr fr1 to1) (Arr fr2 to2)))
      ((union fr1 fr2)
       (union to1 to2)))

;; Simple types
(rewrite (typeof ctx (Num x)) (Int))
(rewrite (typeof ctx (True)) (Bool))
(rewrite (typeof ctx (False)) (Bool))
(rewrite (typeof ctx (Var x))
         (instantiate (lookup ctx x)))

;; Inferring types for lambda abstractions
(rule ((= t (typeof ctx (Abs x e))))
      ((define fresh-tv (TVar (Fresh x)))
       (define scheme (Forall (empty) fresh-tv))
       (define new-ctx (Cons x scheme ctx))
       (define t1 (typeof new-ctx e))
       (union t (Arr fresh-tv t1))))

;; Inferring types for function applcations
(rule ((= t (typeof ctx (App f e))))
    ((define t1 (typeof ctx f))
     (define t2 (typeof ctx e))
     (union t1 (TArr t2 t))))
;; Inferring types for let expressions
(rule ((= t (typeof ctx (Let x e1 e2))))
      ((define t1 (typeof ctx e1 c1))
       (define scheme (generalize ctx t1))
       (define new-ctx (Cons x scheme ctx))
       (define t2 (typeof new-ctx e2))
       (union t t2)))

;; Occurs check
(relation occurs-check (Ident Type))
(relation errors (Ident))
(rule ((= (TVar x) (Arr fr to)))
      ((occurs-check x fr)
       (occurs-check x to)))
(rule ((occurs-check x (Arr fr to)))
      ((occurs-check x fr)
       (occurs-check x to)))
(rule ((occurs-check x (TVar x)))
      ((errors x)
       (panic "occurs check failed")))
```

Fig. 16. Expressing Hindley-Milner inference in egglog. `Ident` is a datatype for identifiers that can be constructed by lifting a string or a counter (i.e., `i64`). In the actual implementation, we additionally track a counter in the `typeof` function to ensure the freshness of fresh variables, which we omit for brevity. Definitions of `instantiate`, `generalize`, and `lookup` are not shown as well.

type and mark types that need to be occurs checked by populating them in the `occurs-check` relation. The occurs check fails when an `occurs-check` demand is populated on an identifier and a type variable with the same identifier. Our actual implementation also contains rules that check if two different base types are unified or a base type is unified with an arrow type, where it will throw an error. These could happen when two incompatible types are unified due to ill-typed terms (e.g., when type inferring `True + 1`).

A.4 Other egglog Pearls

In this subsection, we show more self-contained programs with interesting behaviors, further demonstrating the expressive power of `egglog`.

Equation Solving. Many uses of EqSat and hence `egglog` fall into a guarded rewriting paradigm. A different mode of use is that of equation solving: rather than taking a left hand side and producing a right, `egglog` can take an entire equation and produces a new equation. A common manipulation in algebraic reasoning is to manipulate equations by applying the same operation to both sides. This is often used to isolate variables and use one variable to substitute other variables in an equation. Substitutions in e-graphs and `egglog` are implicit (since the variable and its definition via other variables are in the same equivalence class), and we can encode variable isolation as rules.

```
(datatype Expr
    (Add Expr Expr)
    (Mul Expr Expr)
    (Neg Expr)
    (Num i64)
    (Var String))

;; Algebraic rules over expressions
(rewrite (Add x y) (Add y x))
(rewrite (Add (Add x y) z) (Add x (Add y z)))
(rewrite (Add (Mul y x) (Mul z x))
         (Mul (Add y z) x))

;; Make the implicit coefficient 1 explicit
(rewrite (Var x) (Mul (Num 1) (Var x)))

;; Constant folding
(rewrite (Add (Num x) (Num y)) (Num (+ x y)))
(rewrite (Neg (Num n)) (Num (- n)))
(rewrite (Add (Neg x) x) (Num 0))
```

```
;; Variable isolation by rewriting
;; the entire equation
(rule ((= (Add x y) z))
      ((union (Add z (Neg y)) x)))
(rule ((= (Mul (Num x) y) (Num z))
       (= (% z x) 0))
      ((union (Num (/ z x)) y)))

; system 1: x + 2 = 7
(set (Add (Var "x") (Num 2)) (Num 7))
; system 2: z + y = 6; 2z = y
(set (Add (Var "z") (Var "y")) (Num 6))
(set (Add (Var "z") (Var "z")) (Var "y"))

(run 5) ;; run 5 iterations

(extract (Var "x")) ;; (Num 5)
(extract (Var "y")) ;; (Num 4)
(extract (Var "z")) ;; (Num 2)
```

Fig. 17. Equation Solving in egglog.

Figure 17 shows a simplistic equational system with addition, multiplication, and negations. Besides the standard algebraic rules, we use two rules that manipulate equations to isolate variables. This allows us to solve simple multivariable equations like $\begin{cases} z + y = 6 \\ 2z = y \end{cases}$.

Equation solving in egglog can be seen as similar to the "random walk" approach to variable elimination a student may take. For specific solvable systems this may be very inefficient compared to a dedicated algorithm. For example one can consider a symbolic representation of a linear algebraic system, for which Gaussian elimination will be vastly more efficient. However, equation solving in egglog is compositional and can easily handle the addition of new domain-specific rules like those for trigonometric functions.

Proof Datatypes. Datalog proofs can be internalized as syntax trees inside of egglog. This proof datatype has one constructor for every Datalog rule of the program and records any intermediate information that may be necessary. This can also be done in any Datalog system that supports terms. A unique capability of egglog however is the ability to consider proofs of the same fact to be equivalent, a form of proof irrelevance. This compresses the space used to store the proofs and enhances the termination of the program which would not terminate in ordinary. In addition, the standard extraction procedure can be used to extract a short proof.

Reasoning about matrices. The algebra of matrices follows similar rules as the algebra of simple numbers, except that matrix multiplication generally not commutative. With the addition of structural operations like the Kronecker product, direct sum, and stacking of matrices a richer algebraic structure emerges. A particularly simple and useful rewrite rule allows one to push matrix multiplication through a Kronecker product $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$. Rewriting from left to right improves the asymptotic complexity of evaluating the expression. However, while

```
;; Proofs of connectivity
(datatype Proof
    (Trans i64 Proof)
    (Edge i64 i64))

;; Path function points to a proof datatype
(function path (i64 i64) Proof)
(relation edge (i64 i64))

;; Base case
(rule ((edge x y))
      ((set (path x y) (Edge x y))))

;; Inductive case
(rule ((edge x y) (= p (path y z)))
      ((set (path x z) (Trans x p))))

;; Populate the graph and run
(edge 1 2)
(edge 2 3)
(edge 1 3)
(run)

;; returns the smallest proof of
;; the connectivity between 1 and 3
(extract (path 1 3))
```

Fig. 18. Encoding compact proofs in egglog

```
(datatype MExpr
    (MMul MExpr MExpr)
    (Kron MExpr MExpr)
    (Var String))

(datatype Dim
    (Times Dim Dim)
    (NamedDim String)
    (Lit i64))

(function nrows (MExpr) Dim)
(function ncols (MExpr) Dim)

;; Computing the dimensions of
;; matrix expressions
(rewrite (nrows (Kron A B))
         (Times (nrows A) (nrows B)))
(rewrite (ncols (Kron A B))
         (Times (ncols A) (ncols B)))
(rewrite (nrows (MMul A B)) (nrows A))
(rewrite (ncols (MMul A B)) (ncols B))

;; Reasoning about dimensionality
(rewrite (Times a (Times b c))
         (Times (Times a b) c))
(rewrite (Times (Lit i) (Lit j)) (Lit (* i j)))
(rewrite (Times a b) (Times b a))

;; Rewriting matrix multiplications and Kronecker
;; product
(rewrite (MMul A (MMul B C)) (MMul (MMul A B) C))
(rewrite (MMul (MMul A B) C) (MMul A (MMul B C)))
(rewrite (Kron A (Kron B C)) (Kron (Kron A B) C))
(rewrite (Kron (Kron A B) C) (Kron A (Kron B C)))
(rewrite (Kron (MMul A C) (MMul B D))
         (MMul (Kron A B) (Kron C D)))

;; Optimizing Kronecker product with guarded rules
(rewrite (MMul (Kron A B) (Kron C D))
         (Kron (MMul A C) (MMul B D))
    :when ((= (ncols A) (nrows C))
           (= (ncols B) (nrows D))))
```

Fig. 19. Equality saturation with matrices in egglog. The last rule is guarded by the equational precondition that the dimensionalities should align, which is made possible by rich semantic analyses *a la* Datalog.

this equation may proceed from right to left unconditionally, the left to right application requires that the dimensionality of the matrices line up. In a large matrix expression with possibly abstract dimensionality, this is not easily doable in a classical equality saturation framework. Although one may be tempted to express dimensionality with e-class analyses, the dimensionality is a symbolic term itself and needs to be reasoned about via algebraic rewriting. However, the abstraction of e-class analyses do not allow rewriting over the analysis values. On the other hand, because the analysis is just another function not unlike the constructors for matrix expressions, we can use

standard `egglog` rules to reason about it just like how we reason about matrix expressions. Figure 19 shows a simple theory of matrices with Kronecker product, and this example can be generalized to other (essentially) algebraic theories.

B CORRECTNESS OF THE SEMI-NAÏVE ALGORITHM

In this section, we show the correctness of the semi-naïve algorithm.

Theorem B.1. *The semi-naïve evaluation of an* `egglog` *program produces the same database as the naïve evaluation.*

Proof. We use $I_i^{SN} = (DB_i^{SN}, \equiv_i^{SN})$ and $I_i^N = (DB_i^N, \equiv_i^N)$ to denote the instances produced by the semi-naïve evaluation and naïve evaluation. We prove $I_i^{SN} = I_i^N$ by induction. It is easy to see $I_i^{SN} = I_i^N$ for $i = 0, 1$, and we will prove that, for $i \geq 1$, if $I_j^{SN} = I_j^N$ for $j \leq i$, $I_{i+1}^{SN} = I_{i+1}^N$ holds.

First, because $\Delta DB_i = DB_i^{SN} - DB_{i-1}^{SN} = DB_i^N - DB_{i-1}^N$ by definition, we have

$$DB_{i-1}^N \supseteq DB_i^N - \Delta DB_i, \tag{1}$$

$$T_P(I_{i-1}^N) \supseteq T_P(I_i^N - \Delta DB_i) \quad \text{by monotonicity of } T_P \text{ w.r.t. } \subseteq, \tag{2}$$

$$T_P(I_{i-1}^N) \cup T_P^{SN}(I_i^N, \Delta DB_i) \supseteq T_P(I_i^N - \Delta DB_i) \cup T_P^{SN}(I_i^{SN}, \Delta DB_i), \tag{3}$$

$$\supseteq T_P(I_i^N) \quad \text{by inspecting the definition of } T_P^{SN}. \tag{4}$$

Second, it is straightforward to see that, for all instance I and database DB

$$R^\infty(R^\infty(I) \cup DB) = R^\infty(I \cup DB). \tag{5}$$

Combining these two, we get

$$I_{i+1}^{SN} = R^\infty\left(I_i^{SN} \cup T_P^{SN}(I_i^{SN}, \Delta DB_i)\right), \tag{6}$$

$$= R^\infty\left(\underline{R^\infty\left(I_{i-1}^N \cup T_P(I_{i-1}^N)\right)} \cup T_P^{SN}(I_i^{SN}, \Delta DB_i)\right) \quad \text{by the ind. hypothesis and defn. of } I_i^N, \tag{7}$$

$$= R^\infty\left(\underline{I_{i-1}^N \cup T_P(I_{i-1}^N)} \cup T_P^{SN}(I_i^{SN}, \Delta DB_i)\right) \quad \text{by Eqn. 5}, \tag{8}$$

$$= R^\infty\left(I_{i-1}^N \cup T_P(I_{i-1}^N) \cup T_P^{SN}(I_i^{SN}, \Delta DB_i) \underline{\cup T_P(I_i^N)}\right) \quad \text{by Eqn. 4}, \tag{9}$$

$$= R^\infty\left(I_{i-1}^N \cup T_P(I_{i-1}^N) \cup T_P(I_i^N)\right) \quad \text{since } T_P^{SN}(I_i^{SN}, \Delta DB_i) \text{ is a subset of } T_P(I_i^N), \tag{10}$$

$$= R^\infty\left(\underline{R^\infty\left(I_{i-1}^N \cup T_P(I_{i-1}^N)\right)} \cup T_P(I_i^N)\right) \quad \text{by Eqn. 5}, \tag{11}$$

$$= R^\infty\left(\underline{I_i^N} \cup T_P(I_i^N)\right) = I_{i+1}^N. \tag{12}$$

□

Received 2022-11-10; accepted 2023-03-31