egglog In Practice: Automatically Improving Floating-point Error

OLIVER FLATT, YIHONG ZHANG

Herbie is a tool for automatically improving floating-point accuracy in programs. Egglog is a new language for performing rewriting over equality, supporting robust analysis. In this tutorial, we show how we improved Herbie in two ways. First, we will show how we leverage egglog to perform sound rewriting in the presence of division for Herbie. Second, we show how to use egglog's powerful rules to extract more accurate programs from the database.

1 OVERVIEW

Herbie is a tool for automatically improving floating-point accuracy in programs (Panchekha et al. 2015). A core part of Herbie is using egg, an egraph implementation, to explore equivalent programs (Willsey et al. 2021). It then extracts a sample of these programs, measures their accuracy, and combines them into a final output that has the better accuracy.

However, Herbie suffers a number of problems when using egg. First, Herbie is unable to apply its rewrite rules in a sound way: it frequently uses unsound rules involving division and exponents. Because of the unsound rules, Herbie must take an additional step to validate programs in the egraph and discard bogus programs. Using these rules in a sound way would require proving that the denominator does not error, a complex analysis that is difficult with egg. Second, Herbie's sampling only considers a small number of the equivalent programs in the egraph, so it often misses good alternatives to the original program.

In this tutorial, we will show how we solved both problems using egglog, a language for performing rewriting over equality which supports complex analysis and queries. At the core of egglog, it treats an egraph as a (relational) database and expresses egraph queries and rewrites as database rules. Egglog addresses Herbie's first problem by leveraging functional dependencies, a core concept of egglog, to store complex analysis about the programs in the egraph. By computing an interval analysis in the egraph, we are able to prove that some expressions are non-zero and thus perform rewriting. The second problem is addressed by leveraging egg's powerful query system to compute the error analysis in the database, extracting a most-accurate program at the same time.

2 PROBLEM 1: SOUND REWRITING

2.1 Egglog

Throughout the rest of this tutorial, we will use a fragment of Herbie as the running example. First, let's define a simple language for arithmetic, including numbers, variables, addition, division, and multiplication. The `datatype` macro is used to define a new datatype with several variants.

```
(datatype Math
  (Num Rational)
  (Var String)
  (Add Math Math)
  (Div Math Math)
  (Mul Math Math))
```

Now, let's also define some useful constants.

```
(let zero (Num (rational 0 1)))
```

Author's address: Oliver Flatt, Yihong Zhang.

```
(let one (Num (rational 1 1)))
(let two (Num (rational 2 1)))
```

Next, we can define some rewrites similar to egg.

```
(rewrite (Add a b) (Add b a)) ;; commutativity
(rewrite (Add a zero) a) ;; identity
(rewrite (Add (Num r1) (Num r2)) ;; constant folding addition
         (Num (+ r1 r2)))
```

These rewrites execute as a query over the database of terms, looking for terms that match the pattern on the left. When a match is found, the rewrite unions the result with the right-hand side of the rewrite. These queries are done **modulo equality**, meaning that they perform identically to **e-matching** in an egraph. In other words, they find all matches in the database, sometimes leveraging existing equalities. Actually, the `rewrite` form is just syntactic sugar for a simple `rule` form. The rewrite `(rewrite (Add a b) (Add b a))` is the same as

```
(rule ((Add a b)) ((union (Add a b) (Add b a))))
```

.

One of these rewrites is actually not only syntactic—the constant folding rule performs a primitive addition of rational numbers. We see here the first benefit of egglog over egg: Egg's rewrite language only support syntactic rewrites, and to support rewrites like this require complicated Rust implementations of an analysis or a custom applier. In egglog constant folding can be expressed as a simple rewrite that performs the addition on the right-hand side.

Let's make sure these rules work. In the following program, we define an expression `1+2` and then apply the rewrites to it.

```
(define one-two (Add one two))

(push)
(run 1)
;; yay, constant folding works
(check (= one-two (Num (rational 3 1))))
;; also, commutativity works
(check (= (Add two one) one-two))
(pop)
```

The `push` and `pop` commands, similar to those in SMTLib, push and pop the current state of the database to a stack[1]. We use the `run` command to run all the rules. We then use the `check` command to run a query, throwing an exception if the query does not match.

2.2 Interval Analysis

Let's try to define a more ambitious rule. In math notation, it is written as $\forall x, \frac{x}{x} = 1$. But notice that in egglog notation, the variable we matched on is `x`, but it doesn't appear on the left or right side of the equality. That means we need to union two other terms together, so we can't use the `rewrite` form. In a normal egraph library, this would be a problem, forcing us to use some sort of custom applier for a rule. In egglog, it is simple to define using a rule[2]:

[1] Different from incremental solving in SMTLib, `push` and `pop` in egglog clone the database and are expensive. To implement them efficiently requires support for backtracking.

[2] Actually, this rule doesn't quite work in the current version of egglog, because it doesn't know the type of `x`. This will be fixed in the future.

```
(rule (x)
      ((union (Div x x) one)))
```

The rule matches any `x`, and performs the action of unioning the result of the division with `one`. However, as many readers may guess, this rule has a problem: when `x` is zero, it adds the term $\frac{0}{0}$ to the database. Depending on your semantics, this can cause big problems. In standard math, $\frac{0}{0}$ should certainly be a domain error, which is not the same as `one`.

However, in many cases, it is still sound to use this rule. For example, we can prove that $\sqrt{x}+1$ is either a domain error or it is positive, so we can safely apply the rule. Luckily, in egglog, it is easy to write an interval analysis which tracks the range of values a term can take. In fact, interval analysis is more powerful in the the presence of equality because it can leverage equivalent programs to find a more precise bound (Coward et al. 2022).

First, we define an egglog `function` which stores the interval for every equivalence class.

```
(function ival (Math) Interval
  :merge (intersect old new))
```

A function in egglog is like a relation in a database, but it preserves the functional dependency. That is, at each step there is only one output per input. As such, it needs a way to resolve conflicts when two inputs to the function become equal. For example, if `x + y = y + x` and the function previously had two interval outputs, one for `x + y` and one for `y + x`, then the merge function is called to resolve the conflict. In this case, we simply take the intersection of the two intervals, since the real-valued output must be contained in both intervals.

Now, let's write some rules which compute the intervals for each variant of `Math`.

```
(rule ((Num r))
      ((set (ival (Num r)) (interval r r))))
(rule ((Add a b)
       (= (ival a) i1)
       (= (ival b) i2))
      ((set (ival (Add a b)) (ival-Add i1 i2))))
(rule ((Mul a b)
       (= (ival a) i1)
       (= (ival b) i2))
      ((set (ival (Mul a b)) (ival-Mul i1 i2))))
```

These rules use interval arithmetic primitives defined as a plugin to egglog. They match on variants of `Math`, using the intervals of the subterms to compute the interval of the whole term. The `set` command sets the output of a function to a particular value. Now, we can execute our rule soundly:

```
(rule (x
       (ival-NonZero (ival x)))
      ((union one (Div x x))))
```

The built-in `ival-NonZero` checks that the interval for `x` does not contain zero. When the range does not contain zero, the rule can fire. For example, if we know that a variable `"x"` has the range $[0, 1]$ we can safely apply it to $x + 1$:

```
(let x (Var "x"))
(let x1 (Add x one))

(push)
```

```
(set (ival x) (interval (rational 0 1) (rational 1 1)))

(run 3)

(check (= one (Div x1 x1)))
(pop)
```

2.3 Implementing Intervals in egglog

In the last section, we implement an interval analysis using the interval plugin to egglog implemented in Rust. This gives egglog much power, since we can use state-of-the-art Rust libraries to complement egglog with domain-specific capabilities, but it now requires code both written in egglog and in Rust.

In fact, we don't have to use a Rust plugin if we don't want to. We can define a simple interval library inside egglog. We first define the interval sublanguage as a datatype, which contains both the intervals and operations over them.

```
(datatype Interval
    (I Rational Rational)
    (ival-Add Interval Interval)
    (ival-Mul Interval Interval)
    (intersect Interval Interval))
```

Here is the fun part: we can define the interval semantics as rewrite rules in egglog. For instance, the first rule says, whenever there are terms of the form `(ival-Add (I la ha) (I lb hb))` in the database, egglog should populate `(I (+ la lb) (+ ha hb))` and equate them. We also tag interval-related rules with the `interval` ruleset to separate them from program rewriting rules.

```
(ruleset interval)
(rewrite (ival-Add (I la ha) (I lb hb))
         (I (+ la lb) (+ ha hb)) :ruleset interval)
(rewrite (ival-Mul (I la ha) (I lb hb))
         (I (min (min (* la lb) (* la hb))
                 (min (* ha lb) (* ha hb)))
            (max (max (* la lb) (* la hb))
                 (max (* ha lb) (* ha hb))))
          :ruleset interval)
(rewrite (intersect (I la ha) (I lb hb))
         (I (max la lb) (min ha hb))
         :ruleset interval)
```

Finally, our Herbie demo uses `ival-NonZero` to test whether an interval is non-zero. Here we define `ival-NonZero` as a relation that contains all non-zero intervals in the database.

```
(relation ival-NonZero (Interval))
(rule ((= i (I lo hi))
       (> lo (rational 0 1)))
      ((ival-NonZero i)) :ruleset interval)
(rule ((= i (I lo hi))
       (< hi (rational 0 1)))
      ((ival-NonZero i)) :ruleset interval)
```

During the rewriting, we alternate between the program rewriting and the computation of intervals. This can be done with egglog's schedule, which allows composable and fine-grained rule scheduling:

```
(define-schedule main-schedule (seq
    (run 1) ;; run the default ruleset once
    (saturate interval))) ;; run interval rules to fixpoint
(run main-schedule 3)
```

3 FINDING ACCURATE PROGRAMS

Now that we have a way to rewrite math expressions soundly, let's turn our attention to finding a more accurate program. After all, there are many equivalent programs in the database, and many of them perform very differently in floating-point. But before we can do that, we need to define a way to measure the accuracy of a program.

3.1 Measure Accuracy

In fact, a great way to measure the accuracy of a program on a **particular input** is to use interval arithmetic! By computing an interval arithmetic on a particular point at high precision, we can compute the real value of an expression up to a specified error bound. If our error bound is small enough, we can compute the best possible 64-bit floating-point answer. In fact, this is exactly the technique that Herbie already uses, but it does so outside of the egraph. Here's how it works:

```
;; Set the variable x to a particular input value 200/201
(set (ival x) (interval (rational 200 201) (rational 200 201)))

;; compute the best possible 64-bit floating-point answer
(function true-value (Math) f64)

(rule ((= val (to-f64 (ival expr))))
      ((set (true-value expr) val)))

(run main-schedule 4)
;; prints 1.9950248756218905
(extract (true-value x1))
```

The code sets "x" to a particular value, then runs our same interval analysis rules from before. The interval analysis computes a tight bound on the output value, so we know that 1.9950248756218905 is the best we could do with 64-bit floating-point.

3.2 Finding an Accurate Program

Now that we have a "truth" value for a program on a particular point, we can use it to find a more accurate program. To do this, we write an egglog analysis that keeps track of the most accurate program for each equivalence class. The code below defines a function `best-error` which stores the best floating-point value computed so far. In practice, we would also need a function to store the program itself, but we omit that here for brevity.

```
(function best-error (Math) f64 :merge new :default NAN)

(rule ((Num n))
      ((set (best-error (Num n)) (to-f64 n))))
```

The code uses a `merge` function which always accepts the new value. In general, the merge function needs to be monotonic, but in this case it is sound because our rules will only populate the table with values that converge on the "truth" value. We also include a *:default* option which specifies the default value for the function.

Finally, we can write a rule that computes the best error for a particular program, using the best error for its subterms. Importantly, the rule needs to only fire when the error is better than the current best error. Here, we use the *rel-error* built-in function to compare to the *true-v*, the "truth" value for the program.

```
(rule ((= expr (Add a b))
       (= (best-error a) va)
       (= (best-error b) vb)
       (= true-v (true-value (Add a b)))
       (= computed (f64-Add va vb))
       (< (rel-error (best-error (Add a b)) true-v)
          (rel-error computed true-v)))
      ((set (best-error (Add a b)) computed)))
```

Unfortunately, this method cannot find us the optimal program for the input, because floating-point accuracy does not always increase monotonically. It could be the case that the best program for the input has higher floating-point error at an intermediate term, but this error is cancelled out later during computation. However, monotonicity is a good approximation for the optimal output in practice. In any case, this procedure is certainly better than Herbie's old sampling-based approach.

BIBLIOGRAPHY

Samuel Coward, George A. Constantinides, and Theo Drane. Abstract Interpretation on E-Graphs. 2022.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50(6), 2015. https://doi.org/10.1145/2813885.2737959

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