A Trick that Makes Classical E-Matching Faster

Yihong Zhang

You might have seen our POPL 2022 paper on Relational E-matching, where we use database techniques to improve an important procedure in e-graphs, namely e-matching. We made e-matching orders of magnitude faster, proved theoretical bounds of e-matching, and opened the door for all kinds of wild things you can do with databases and e-graphs. I'm very proud of this work, not only because it is elegant, fast, and theoretically (worst-case) optimal, but also because it is the kind of work I'd like to work on: building connections across areas.

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

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

1. start a clean-slate relational e-graph framework that handles all e-graph operations efficiently and forget about the graph part of an e-graph.
2. keep the current e-graph data structure, and port some of the good bits of relational e-matching back to it.

I will focus on the second approach (and save the first approach for a future blog post). When working on relational e-matching, we found an optimization to the backtracking-style classical e-matching. Like relational e-matching, it is able to improve e-matching asymptotically in some cases, but it does not require transforming the input to a relational database. And it is very simple. For what it's worth, this optimization (instead of relational e-matching) is what is currently implemented in egg.

In this blog post, I will describe this optimization. But before that, let's have some brief background on e-matching. Feel free to skip you already know what e-matching is.

E-Matching

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

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

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

The optimization

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

Let's first look at the classical e-matching algorithm (Fig. 3 of the relational e-matching paper, with a typo fix).

$$\begin{aligned}
\mathit{match}(x, c, S) =& \{\sigma \cup \{x \mapsto c\} \mid \sigma \in S, x \notin \mathrm{dom}(\sigma)\} \cup \\
& \{\sigma \mid \sigma \in S, \sigma(x) = c\} \\
\mathit{match}(f(p_1, \ldots, p_k), c, S) =& \bigcup_{f(c_1,\ldots,c_k)\in c} \mathit{match}(p_k, c_k, \ldots, \mathit{match}(p_1, c_1, S))
\end{aligned}$$

It takes a pattern p, an e-class c, and current substitutions S, and returns the set of substitutions produced by e-matching p over e-class c, such that all produced substitutions are extensions of some substitutions in S. The result of e-matching a pattern p over an e-graph is $\bigcup_{c\in C} \mathit{match}(p, c, \{\emptyset\})$ (both our paper and the paper our definition is based on have another typo here), where C is the set of e-classes in the e-graph.

The algorithm is straightforward:

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

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

To implement this idea, we lift case 1.b to the top-level of the algorithm. During e-matching, the algorithm will first check whether the free vars of the input pattern is contained in the domain of the substitution. If yes, then instead of looking further into the pattern, the algorithm will lookup the substituted term for comparison. The following definition shows this:

$$\begin{aligned}
\mathit{match}(p, c, S) =& \begin{cases} \{\sigma \mid \sigma \in S, \mathit{lookup}([\sigma]e) = c\} & \text{if } \mathrm{fv}(p) \subseteq \mathrm{dom}(S) \\ \mathit{match}'(p, c, S) & \text{o.w.} \end{cases} \\
\mathit{match'}(x, c, S) =& \{\sigma \cup \{x \mapsto c\} \mid \sigma \in S\} \\
\mathit{match'}(f(p_1, \ldots, p_k), c, S) =& \bigcup_{f(c_1,\ldots,c_k)\in c} \mathit{match}(p_k, c_k, \ldots, \mathit{match}(p_1, c_1, S))
\end{aligned}$$

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

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

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

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

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

A Relational View of the Trick

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

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

In the relational e-matching paper, we also described how we use functional dependency to speed up queries. In fact, if the variable ordering of generic joins follows the topological order of the (acyclic) functional dependency, the run-time complexity will be worst-case optimal under the presence of FDs, a stronger guarantee than the original AGM bound. Functional dependencies are also exploited for query optimization in databases.

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

Query planning

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

If we assume the cost of enumerating each node is the same, this problem can be viewed as finding the smallest connected component (CC) in the pattern tree that contains the root, such that the CC covers all distinct variables. This does not sound like an easy problem. I currently have two ways for solving this problem in my mind: 1) do a dynamic programming on trees with exponential states, and 2) reduce it to an ILP problem. Both sound like overkill for realistic queries though.

I'm not sure what is a practically good planning heuristic. The one used in egg prioritizes sub-patterns with more free vars, but I'm skeptical how good it is: consider pattern $f(f(g(\alpha), \beta)), g(h(\alpha), h(\beta)))$. This heuristics yield the following plan for this pattern:

```
for f1 in c: # f(f(g(a), g(b))), g(h(a), h(b)))
  for f2 in c.child1: # f(g(a), g(b))) (2 free vars)
    for g1 in c.child2: # g(h(a), h(b)) (2 free vars)
      for g2 in f2.child1: # g(a) (1 free var)
        h1 = lookup(mk_node(h, g2.child1) # lookup h(a)
        if h1 is None or h1 != g1.child1:
```

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

This is complicated, but it suffices to only look at the first three loops: It does a cross product over the first and the second child of the top-level f-node. It seems a good strategy is instead to prefer fewer free vars, and performs the search in a depth-first search, so that $g(h(a), h(b))$ can be looked up all at once after $f(g(a), g(b))$ is enumerated. But is preferring fewer free vars the right way to go? I don't know. Also realistic patterns I've seen so far tend to be small and simple, so cases like the above may be rare.

Miscellaneous

I thank Max and Philip for their valuable discussions and comments. The presented trick stems from a PR that tries to improve e-matching for ground terms. In hindsight, the proposed improvement had been discussed in Leo's e-matching virtual machine paper but was lost in egg's original presentation. While the PR only looks up ground terms, this optimization generalizes it by also looking up terms that are grounded after substitution. Philip came up with this idea independently as well.