E-graph theory

Sofia M.A.

E-graphs

Given the algorithms in Willsey et al. seem to be wrong, this is an attempt at writing the clearest algorithmic exposition I can

We must make sure to distinguish e-classes from e-nodes, and canonical e-class IDs from non-canonical! Let's call non-canonical e-class IDs `Pointer`s and canonical e-class IDs `Ref`s

So, a canonicalized e-node is some node with e-classes as its children: `f Ref` for some `f`. And a canonical e-graph consists of the following structure:

- a mapping from `Ref`s to canonical e-classes. Call this `classes`

where a canonical e-class is

- A set of `f Ref`

Now, two `Ref`s are equivalent if they map to the same e-class. That's it!

In order to allow intermediate states, things become more complex.

- we introduce a union-find data structure that is the source of truth for when two e-classes are the same. This gives us three operations: `ufFind`, `ufInsert`, and `ufUnion`.
- we add an inverse mapping from `f Ref`s to `Ref`s, to allow easy lookup of canonicalized e-nodes. Call this `share`

Now, the three processes appear as follows

```
find : Pointer → Ref
find = ufFind

insert : f Ref → { Ref, new: Bool }
insert f =
    if f is in share
    then share[f]
    otherwise
        r ← ufInsert
        share[f] ← r
```

```
        classes[r] = {f}
        r

# returns the new Ref for the unioned class
union : Ref → Ref → Maybe Ref
union a b =
    if a = b
    then Nothing
    otherwise
        top ← ufUnion a b
        classes[top] = classes[a] ∪ classes[b]
        apply substitution [a := top, b := top] for every `f Ref` in share
            if any two `f Ref`s become equal, union their corresponding Refs
        apply substitution [a := top, b := top] for every `f Ref` in every class in classes
        Just top
```

Union is very expensive. To fix this, we apply a little denormalization: * `share`, mapping `f Ref`s to `Ref`s, can simply map them to `Pointer`s instead and use `find` where appropriate to canonicalize again. The motto: we should generally use `Pointer`s and not `Ref`s in covariant positions. * Applying this to `classes`, we can have a list of `f Pointer`s instead, at the risk of maybe h * To each e-class, we add a list of parents: pairs of (f Ref, Ref). Now, instead of going through everything in `share` and `classes`, we can limit our search to what is necessary:

```
find : Pointer → Ref
find = ufFind

canonicalize : f Pointer → f Ref
canonicalize = traverse find

insert : f Ref → { Ref, new: Bool }
insert f =
    if f is in share
    then find(share[f])
    otherwise
        r ← ufInsert
        share[f] ← r
        classes[r] = {(f, {})}
        for i in f:
            parents[i] = parents[i] ∪ {(f, r)}
        r

# returns the new Ref for the unioned class
union : Ref → Ref → Maybe Ref
union a b =
    if a = b
```

```
    then Nothing
    otherwise
        top ← ufUnion a b
        classes[top] = classes[a] ∪ classes[b]
        new_parents = parents[a] ∪ parents[b]

        for (f, r) in new_parents:
            remove share[f]
            share[canonicalize(f)] = find(r)
            if canonicalize(f) in parents[top]
                union(r, parents[top][canonicalize(f)])
            parents[top] = parents[top] ∪ {(canonicalize(f), find(r))}

        # this is no longer necessary, as we don't canonicalize e-nodes within e-classes
        # for class in parents[top]:
        #   for node in class:
        #       remove node from class
        #       insert canonicalize(node) into class

        Just top
```

Now, we can simply push the `new_parents` set onto a worklist and defer the loop in union. As far as I can tell, this is more-or-less the approach taken by the `egg` library