A Generic, Anytime Equality Saturation Algorithm

Remy Goldschmidt
regolds2@illinois.edu
University of Illinois at Urbana-Champaign

May 16, 2018

Contents

1 Introduction

Equality saturation is a framework for optimization first introduced in a 2009 POPL paper [12] by Tate et al. The *phase ordering problem* in compiler optimization is essentially the issue of figuring out in what order optimizations should be applied to code; this problem is very difficult because some optimizations expose code that allows other optimizations to be applied, while other optimizations remove opportunities to apply optimizations. Equality saturation solves the phase ordering problem by restructuring optimization as saturation-based automated theorem proving ("forward chaining") followed by combinatorial optimization.

The equality saturation architecture described in this paper is currently being implemented in an open source Apache 2.0 licensed project hosted on GitHub: `https://github.com/taktoa/eqsat`.

Note that although we often use the Haskell list type constructor `[]` in this paper, a real implementation of equality saturation would likely use `Vector` or `Seq` from the `vector` and `containers` libraries respectively. In general, this paper will focus on describing algorithms in a specific-but-unperformant way, followed by a vague-but-performant descripton.

2 | Abstract equality saturation

The basic outline of equality saturation is that the user must first convert a piece of code (usually a control flow graph) into a referentially transparent directed graph with sharing, which is called a *program expression graph* (PEG)[i]. Here, "referentially transparent" means that, assuming there is a transition system $(S, \to)$ representing the semantics of the language, the semantics of a PEG node are defined purely by the node label and the semantics of the children of that node (its "out-neighbors"). In addition to a PEG, equality saturation has two other inputs:

1. A term-rewriting system on PEGs, whose rules define the basic optimizations that the equality saturation engine will compose together. For example, this term-rewriting system could simply be the operational semantics TRS of the programming language, in terms of PEGs (using this TRS would make the optimizer quite similar to a partial evaluator).
2. A heuristic for the runtime performance of a given PEG. In the most general case, this is simply a function of type `Term -> R` for some partially ordered semiring `R`, which is usually an integer or floating-point type but may also be something more sophisticated like the symbolic integer types in the `sbv` package.

The output of equality saturation is an optimized PEG, which can then be turned back into a control flow graph for code generation. Since equality saturation involves a time-consuming breadth-first equational proof search, there is also an "anytime" variant of equality saturation where the best known PEG so far is emitted every so often (a user-specified *timer* of type `IO Bool` is called every time a rule is applied, and if it returns `True`, the best PEG found so far is passed to a user-specified *callback*).

Consider the types described in Listing 2.1. This is the most basic abstract description of equality saturation. The intended semantics of the `saturate` function are the following:

1. Convert the given term to a PEG, and then to an EPEG.
2. Apply a rule to a node.
3. Call the "timer":
   (a) If it returns `True`, select the best sub-PEG using the heuristic and call the callback with this PEG (and its quality and equality proof).
   (b) Otherwise, go to step 2.

[i] Note that the way PEGs are defined in [12] is actually more specific (due to the details of optimizing imperative languages) than the way I will use the term in this paper; this is one of the ways in which this description of equality saturation is more generic than previous expositions.

Before we can discuss the laws of the `saturate` function, we must first define some terminology:

- Define the *application of an `Equation` to a `Term` focused at a `TermCursor`* to be the result of running the following algorithm:

```
applyFocused :: Equation n -> Term n -> TermCursor -> Maybe (Term n)
applyFocused e = go
  where
    go :: Term n -> TermCursor -> Maybe (Term n)
    go t                    [] = rewrite e t
    go (Node (n, cs)) (i : tc) = do cs' <- iforM cs (\j c -> if i == j
                                                             then go c tc
                                                             else pure c)
                                    pure Node (n, cs')
    go _                    _ = Nothing

    iforM :: (Monad m) => [a] -> (Int -> a -> m b) -> m [b]
    iforM xs f = mapM (uncurry f) (zip [0 ..] xs)
```

  where `rewrite :: Equation n -> Term n -> Maybe (Term n)` is the function that rewrites a `Term` using an `Equation` and returns the rewritten `Term` iff the left-hand-side of the given `Equation` unifies with the given `Term`.

- Define the *application of a `Proof` to a `Term` in an `LTRS`* to be the result of sequentially applying each rule obtained by looking up the preimage of the label in that `ApplyRule` node to the `Term`, focused at the `TermCursor` of that `ApplyRule` node. In code, this looks like:

```
applyProof :: LTRS n l -> Proof l -> Term n -> Maybe (Term n)
applyProof ltrs = flip go
  where
    go :: Term n -> Proof l -> Maybe (Term n)
    go t = \case
      QED -> pure t
      (ApplyRule tc l rest) -> do
        e <- equationFromLabel ltrs l
        t' <- applyFocused e t tc
        go t' rest
```

  where `equationFromLabel :: LTRS n l -> l -> Maybe (Equation n)` is the function that looks up the preimage of a label in an `LTRS`.

- For most monads `m`, the denotation of a value `timer :: Timer m` is a function $[\![$`timer`$]\!]$ of type `Context m -> Bool`, where `Context m` is basically a data family. In principle, execution of the timer function could reasonably cause side effects, but we will assume that these side effects cannot affect the semantics of any of the code we are dealing with. If $\texttt{timer}_1$ `:: Timer m` and $\texttt{timer}_2$ `:: Timer m`, then we say that $\texttt{timer}_1 \subseteq \texttt{timer}_2$ iff for every `c :: Context m`, $[\![\texttt{timer}_2]\!]$ `c` being `True` implies that $[\![\texttt{timer}_1]\!]$ `c` is `True`.

Now we can state the informally-defined laws of the `saturate` function as follows:

- Assume that the following terms exist in scope: `ltrs :: LTRS n l`, `h :: Heuristic m n r`, $\texttt{t}_i$ `:: Term n`, and `cb :: Callback m n r l`.
- Suppose we call `saturate ltrs h` $\texttt{t}_i$ `(pure True) cb`.

  Define `calls :: [(Term n, r, Proof l)]` to be the list containing, from earliest to latest, the argument of every call to `cb` at an arbitrary point in program execution.
- For every element `(t, q, p) :: (Term n, r, Proof l)` of `calls`, the following properties must hold:
  1. It must be the case that `t` is in the reflexive-transitive closure of the term-rewriting system given by `ltrs` starting at $\texttt{t}_i$.
  2. It must be the case that applying `p` to $\texttt{t}_i$ yields `t`.
  3. It must be the case that if `cb (t, q, p) :: m Bool` evaluates to `True`, then `(t, q, p)` is the last element of `calls`.
- Now assume that $\texttt{c}_1$ `:: (Term n, r, Proof l)` and $\texttt{c}_2$ `:: (Term n, r, Proof l)` are arbitrary elements of `calls` such that $\texttt{c}_1$ occurs before $\texttt{c}_2$, and define $\texttt{(t}_1\texttt{, q}_1\texttt{, p}_1\texttt{) = c}_1$ and $\texttt{(t}_2\texttt{, q}_2\texttt{, p}_2\texttt{) = c}_2$. Then $\texttt{q}_1 < \texttt{q}_2$, where $\leq$ is the partial order defined on the `r` type.
- If `ltrs` is confluent and terminating, then the length of `calls` must be finite.
- Suppose we have $\texttt{timer}_1$ and $\texttt{timer}_2$ `:: Timer m` and we define $\texttt{calls}_1$ and $\texttt{calls}_2$ to be values of type `[(Term n, r, Proof l)]` containing, from earliest to latest, the argument of every call to `cb` during the execution of `saturate ltrs h` $\texttt{t}_i$ $\texttt{timer}_1$ `cb` and `saturate ltrs h` $\texttt{t}_i$ $\texttt{timer}_2$ `cb` respectively.

  Then $\texttt{timer}_1 \subseteq \texttt{timer}_2$ must imply that $\texttt{calls}_1$ is a subsequence of $\texttt{calls}_2$.

```haskell
type Variable = Natural

data TermRepr = G | T

data Term (repr :: TermRepr) (node :: *) (var :: *) where
  Ref  :: Natural                      -> Term 'G   node var
  Var  :: var                          -> Term repr node var
  Node :: (node, [Term repr node var]) -> Term repr node var

type OpenTTerm   node = Term 'T node Variable
type OpenGTerm   node = Term 'G node Variable
type ClosedTTerm node = Term 'T node Void
type ClosedGTerm node = Term 'G node Void

-- Any Term can be converted to a GTerm, but not vice versa.
upcastTerm :: Term repr node var -> GTerm node var

-- Invariant: variables in right term are a subset of variables in left term
type Equation node = (TTerm node Variable, GTerm node Variable)

type TRS  node       = Set (Equation node)
type LTRS node label = InjectiveMap (Equation node) label

type TermCursor = [Natural]

data Proof label = QED | ApplyRule TermCursor label (Proof label)

type Heuristic m node real = (Term node -> m real)

type Timer m = m Bool

type Callback m node real label
  = ( Term node   -- The best term discovered so far
    , real        -- The quality of this term
    , Proof label -- A proof of equality with the original term
    ) -> m Bool   -- Returning False here will quit saturation

saturate :: (Ord real, Monad m)
         => LTRS node label            -- The rewrite rules to optimize with
         -> Heuristic m node real      -- The performance heuristic
         -> Term node                  -- The term to optimize
         -> Timer m                    -- The "timer"
         -> Callback m node real label -- The "callback"
         -> m ()
```

Listing 2.1: The most basic exposition of equality saturation

3 | Improving performance

It turns out that the naïve implementation of the abstract interface described in chapter 2 has a variety of performance problems, which we will address in this chapter one-by-one.

3.1 Avoiding intermediate data structures

The algorithm described in Listing 2.1 has the `Callback` type accepting a `Term`. Unfortunately, this means that the relevant sub-PEG must entirely be copied every time the callback is called, and any sharing in the PEG will be lost after this copy is done. We can solve this by defining the `Callback` type like

```
class (Monad m) => IsTerm (m :: * -> *) (term :: * -> *) where
  getContents :: term n -> m n
  forChildren :: term n -> (term n -> m void) -> m ()

getChildren :: (IsTerm (ST s) term) => term n -> ST s [term n]
getChildren term = do
  temp <- newSTRef []
  forChildren term (\child -> modifySTRef' (child :) temp)
  readSTRef temp

instance (Monad m) => IsTerm m ClosedTTerm where
  getContents (Node (n,        _)) = pure n
  forChildren (Node (_, children)) = mapM_ f children

data Callback (m :: *) (node :: *) (real :: *) (label :: *) :: * where
  Callback :: (IsTerm m term)
           => ((term node, real, Proof label) -> m Bool)
           -> Callback m node real label
```

so that it wraps up an `IsTerm` dictionary, rather than requiring a concrete `Term` value. We can then define an instance of `IsTerm` on a (rooted) PEG type, so that `saturate` can instantiate the existentially quantified type variable bound by the `Callback` constructor to that type.

The reason `IsTerm` has an extra type parameter corresponding to the ambient monad is to account for the possibility that the type we are using for PEGs requires a constraint on the ambient monad (e.g.: since a PEG is mutable it will likely want a `m ~ ST s` or `m ~ IO` or `PrimMonad m` constraint, where `PrimMonad` is a typeclass defined in the `primitive` package encompassing monads like `ST s` and `IO` and `ReaderT Foo IO` that allow mutation).

3.2 Automatically-applied rules

In some cases, we may have knowledge that a rule is *always* an improvement, regardless of the assignment of terms to its metavariables. For example, if our cost heuristic is expressed symbolically as a function that can be sent to an SMT solver, then for each rule we may try using the SMT solver to prove that there is an assignment that makes the cost heuristic worse after applying the rule. Then if the SMT solver returns `unsat`, we know that the rule is always an improvement.

If we have this knowledge, then it makes sense to make these rules *automatic*, in the sense that they are always applied whenever possible, before any other rules. Of course, this makes our optimization algorithm greedy, since the application of these automatic rules may prevent other rules from being applied (and these other rules may be better optimizations). It may be possible to avoid this greediness by also accepting a proof that the rule doesn't affect the set of applicable optimizations, but I suspect that this constraint will be so strong that it excludes nearly all of the relevant rules. Finding the least-restrictive constraint on rules for automatic application is an interesting subject for future research.

3.3 Maximal sharing

The version of equality saturation I have shown you thus far has a major issue: it doesn't account for sharing in the EPEG. We can solve this via hash-consing, although the presence of cycles in the EPEG presents a problem for most hash-consing algorithms. The solution is to compute the set of strongly-connected components (SCCs) of the EPEG, and then hash-cons the graph the nodes of which are pairs of EPEG nodes and the SCCs they belong to and the edges of which are a spanning tree of the EPEG. Pairing the nodes with their SCCs ensures that the cycles are included in the hash of each node, and an EPEG can be recovered by reintroducing the cycles from the SCCs.

3.4 Cycle detection

Consider any term rewriting system with a rule like $f(x, y) \mapsto f(y, x)$ for some function symbol f; it is clear that such a rewrite rule will cause a naïve implementation of equality saturation to run forever on any term including f, since the rule can always be applied. In many cases we can prevent such loops in the following way:

- If there are n rules in the term rewriting system, associate a bitvector of length n, filled with zeros, to each node in the EPEG. For a node A, we will denote this bitvector by $v_A \in \{0, \ldots, n-1\} \to \mathbf{2}$.
- When we merge nodes A and B during hash-consing into a node C, it should be the case that $v_C = \vee_{\mathsf{bitwise}}(v_A, v_B) = \{(i, v_A(i) \vee v_B(i)) \mid i \in \{0, \ldots, n-1\}\}$.
- If the ith rule in the term rewriting system is going to be applied to node A, first check whether $v_A(i) = 1$. If so, then skip applying this rule. Otherwise, set the value of $v_A(i)$ to 1 and apply the rule.

Unfortunately, as the number of rules increases, this solution becomes slower and slower, as the bitvectors will tend to be sparse (mostly zeros). Ideally, an adaptive data structure would be used that is the same as a bitvector as long as the size of the bitvector is smaller than that of a cache line, and once this limit is reached, it becomes a trie or hashset. Since the number of rules is completely known as soon as equality saturation begins, such an adaptive data structure need not have any runtime overhead relative to its underlying data structures.

3.5 Online SCC computation

The sharing algorithm described in § 3.3 requires recomputation of the strongly-connected component graph of the EPEG every time a node is added. This is inefficient, since most of the work done in each SCC computation will be similar before and after the addition of a node. This inefficiency can be prevented by using any of the online algorithms for SCC computation. As of this writing, the state of the art for online SCC computation is described in a 2015 paper [1] by Bender et al. There are two algorithms defined in the paper by Bender: one that is optimal for sparse graphs, and one that is optimal for dense graphs. It is not clear to me which algorithm is more relevant to equality saturation, since although it seems at first glance that our graphs are sparse, the whole point of an EPEG is to maximize sharing, so it may be that the graphs end up being dense. This will probably require empirical study; it is likely that constant factors are a bigger concern than the asymptotics of these two algorithms.

3.6 Term indexing

Term indexing is a field of study in which the problem of compactly storing a large number of open terms such that the index can be queried for a subset that includes the set of added terms that will unify with a query term [9]. A term index is best described by the typeclass shown in Listing 3.1.

A term index effectively acts as a filter that can be applied before matching; it allows false positives, but does not allow false negatives, much like a Bloom filter [2]. By filtering the set of patterns before we do pattern matching, we can decrease the amount of work that needs to be done during pattern matching. A term index is a *perfect filter* if it never returns a false positive; otherwise it is an *imperfect filter*.

Every `TermIndex` has a mutable (`Mut`) version and an immutable version. The `freeze` and `thaw` methods allow interconversion between these types. The `empty` method is the only way to create a term index from nothing. The laws of the `TermIndex` typeclass are:

$$\texttt{thaw} \gg\!= \texttt{freeze} = \texttt{pure} \tag{3.1}$$
$$\texttt{insertMany}\ i\ [] = \texttt{pure}\ () \tag{3.2}$$
$$\texttt{insertMany}\ i\ (x \mathbin{++} y) = (\texttt{insertMany}\ i\ x \gg \texttt{insertMany}\ i\ y) \tag{3.3}$$
$$\texttt{queryMany}\ i\ [] = \texttt{pure}\ () \tag{3.4}$$
$$\texttt{queryMany}\ i\ (x \mathbin{++} y) = (\texttt{queryMany}\ i\ x \gg \texttt{queryMany}\ i\ y) \tag{3.5}$$
$$\texttt{queryMany empty}\ q = \texttt{pure}\ () \tag{3.6}$$
$$\texttt{queryManyMut}\ i_m\ q = (\texttt{freeze}\ i_m \gg\!= (\lambda i \mapsto \texttt{queryMany}\ i\ q)) \tag{3.7}$$

The reason I define the fundamental insertion and querying operations of a `TermIndex` as `insertMany` and `queryMany`, rather than as

```
class TermIndex (index :: * -> * -> * -> *) where
  -- ... other definitions ...
  insert :: (Key n v) => Mut index n v val s -> TTerm n v -> val -> ST s ()
  query  :: (Key n v, Monad m) => index n v val -> TTerm n v -> m [val]
```

is for two reasons. Firstly, there are some term indexing data structures that are more efficient when you are inserting or querying multiple terms at the same time, and since `insert` and `query` can be defined in terms of `insertMany` and `queryMany` relatively easily, it makes more sense for `insertMany` and `queryMany` to be the primitive operations. Secondly, by having `queryMany` accept a monadic callback rather than returning a list of results, we can avoid the creation of intermediate data structures.

While it is possible to define `insert` and `query` in terms of `insertMany` and `queryMany` respectively, it causes an additional allocation and pointer indirection for the singleton list, so in an actual version of the `TermIndex` typeclass, `insert` and `query` would be defined as methods with default implementations. For additional ease of defining instances, `insertMany` and `queryMany` can also have default implementations in terms of `insert` and `query` respectively, and to avoid people writing empty instances (which would loop infinitely), a `MINIMAL` pragma can be added, which will allow GHC to check that a valid set of methods have been defined.

3.6.1 Discrimination trees

A *discrimination tree* is a term-indexing data structure based on a trie [9]. Define the *path* of a term `t :: TTerm n v` to be a value of type `[Maybe n]` representing the preorder traversal of the term, replacing any variable nodes with `Nothing`. Inserting a term-value-pair into a discrimination tree is the same as inserting the path computed from the term along with the value into a trie of type `Trie (Maybe n) val`. Querying a discrimination tree for a given term `q :: TTerm n v` is the same as computing the path of `q` and doing a backtracking search through the trie for matching paths; this is much the same as the traditional trie retrieval algorithm, except that when the `Maybe n` value you're searching for at a node is `Nothing`, you nondeterministically search in all the the children, and when the value you're searching for at a node is `Just n`, you nondeterministically search in the `n` and `Nothing` children.

Note that discrimination trees only work when your terms don't include variadic function symbols. If they do include variadic function symbols, you can still use discrimination trees by simply augmenting your node type with the number of children at that node, effectively introducing one function symbol for every possible variadic usage.

For cache efficiency, it is ideal to implement a discrimination tree using a trie based on a contiguous growable array (and the children of each node should be referenced with `Word32` offsets rather than pointers, since in practice the tree will rarely exceed 4 GiB).

3.6.2 Substitution trees

A *substitution tree* is a term-indexing data structure based on a tree of substitutions with values at leaf nodes [5, 9]. The intended semantics of such a tree is that each substitution in composed with the substitutions above it to compute a filter for terms.

```
type Sub node var = Map var (TTerm node var)

domain :: Sub node var -> Set var

type Indicator = Int

data SubTree node var value
  = Leaf value
  | Branch (Sub node (Either Indicator var)) [SubTree node var value]
```

There are three invariants that must be true for the `SubTree` type, which may be enforced by making `SubTree` an abstract type in its own module. Firstly, if the size of the list of children at a `Branch` node is $n \in \mathbb{N}$, then it must always be true that $n \neq 1$. Secondly, for every path `[(σ₁, ω₁), …, (σₙ, ωₙ)] :: [Sub n (Either Indicator v), [SubTree n v val]]` from the root to a leaf of a non-empty tree it must be the case that the variables introduced by the composition `ωₙ` $\circ \ldots \circ$ `ω₁` are all `Left i` for some `i :: Indicator`. Thirdly, for every such path and every $j \in \{1, \ldots, n-1\} \subset \mathbb{N}$, it must be the case that, when we define $k = j + 1$, `null (domain σₖ ∩ (domain σ₀ ∪ … ∪ domain σⱼ)) == True`.

Refer to [5] for a description of the insertion and querying algorithms on substitution trees.

3.7 Caching performance heuristic executions

One of the disadvantages of the version of equality saturation described thus far is that the heuristic must be run on every node of the EPEG every time the timer returns `True`. For a large class of heuristics, it turns out that this is not actually necessary. Since the EPEG is referentially transparent, any heuristic that is definable solely in terms of the semantics of a node can be expressed as a catamorphism of type `(node, [real]) -> real`. A sufficient but not necessary condition for a heuristic being definable in this way is if the definition of semantics is purely operational in nature; this is true in most strictly evaluated languages but not in many lazy languages.

If a heuristic is definable as a catamorphism, then we can cache heuristic executions by noticing that any subtree of the EPEG that is unchanged between the current EPEG and the last time the best sub-PEG was selected will continue to have the same best sub-PEG.

We can take this further, however. If we know that we have saturated a subtree with equalities (e.g.: since the cycle-detection algorithm described in § 3.4 collects information on what rules have been applied to a given node, we can propagate a token that represents the fact that a given subtree is saturated), then we can entirely throw out ("garbage collect") all the other (non-optimal) sub-PEGs in that subtree. Note that when we throw away the non-optimal sub-PEGs, we don't reset the cycle-detection data structures associated with each node, thereby preventing the saturation algorithm from exploring already-explored parts of the search space.

There is a slight issue with this "garbage collection" strategy. Suppose that we garbage collect some nodes, and then another part of the EPEG ends up being rewritten to a term that would have gotten shared with the garbage collected nodes; this will cause the saturation algorithm to repeat work it has already done! To prevent this, we can "axiomatize" the optimizations: for each element of the equivalence class of terms generated by the sub-EPEG we are "garbage collecting", we will introduce a (derivable) rewrite rule with ground terms on both sides to the term-rewriting system used for equality saturation. This added rewrite rule will have to be specially marked so that the cycle-detection data structure gets correctly initialized after applying it; since we know that the result of that rewrite rule is already the best in its equivalence class, we don't want to allow rewriting out of the result (e.g.: if a bitvector is used for cycle-detection, it should be filled with 1s rather than the usual 0s). Of course, even with the space savings given by our term index, these added axioms will take up at least as much space as the nodes we just garbage collected; the goal here is actually to eliminate duplication of edges relating to proofs of equality and wasted time during combinatorial optimization. This garbage collection idea also prevents equality saturation from being used for translation validation, so it is perhaps not worth implementing.

It is unclear to me whether any of these ideas will actually create speed improvements in practice, but they are probably worth exploring.

3.8 Parallelization

Since equality saturation is, at its heart, a breadth-first search, it seems quite amenable to parallelization. However, the maximal sharing algorithm described in § 3.3 creates a problem; worker threads cannot work independently on different parts of the graph without mutexes if sharing might combine these independent sections. There are several possible solutions to this, but the simplest to implement is that the maximal sharing algorithm should run independently of the worker threads, locking the entire graph while deduplication occurs. In this exposition of equality saturation, we need to freeze the entire graph anyway to capture the corresponding `TTerm` to be passed to the callback, so this seems like not a huge loss, but there may be a way to implement parallel anytime equality saturation with less time devoted to blocking.

```haskell
type Key node var = (Ord node, Ord var)
class TermIndex (index :: * -> * -> * -> *) where
  -- An injective type family defining the mutable version of an immutable
  -- term index. The extra s parameter is for an ST-style state token.
  type Mut index (node :: *) (var :: *) (value :: *) (s :: *)
    = (result :: *) | result -> index

  empty        :: index node var value
  freeze       :: Mut index node var value s
               -> ST s (index node var value)
  thaw         :: index node var value
               -> ST s (Mut index node var value s)
  insertMany   :: (Key node var)
               => Mut index node var value s
               -> [(TTerm node var, value)]
               -> ST s ()
  queryMany    :: (Key node var, Monad m)
               => index node var value
               -> [(TTerm node var, value -> m any)]
               -> m ()
  queryManyMut :: (Key node var)
               => Mut index node var value s
               -> [(TTerm node var, value -> ST s any)]
               -> ST s ()
```

Listing 3.1: A typeclass describing a term index

4 | Improving expressiveness

In chapter 3 we discussed ways to improve the performance of equality saturation. In this chapter, we will discuss ways to improve the *expressiveness* of equality saturation, in two senses: some of these improvements will make the set of languages optimizable with a generic equality saturation implementation larger, while the rest will simply make it easier to use the equality saturation implementation.

4.1 Variadic function symbols

Many languages contain constructors that are in some sense variadic. This variadicity can be eliminated by adding a constructor to the `Term` type that allows for variadic function symbols. This would allow term indexing data structures like discrimination trees, which do not ordinarily handle variadic function symbols, to be expressed in a more type-safe way.

4.2 Adding equational axioms

Many languages contain constructors that should be considered equal under some non-syntactic equivalence relation. For example, if you have multiple independent modules in the code you are optimizing, then cross-module inlining is only possible if the constructor for a set of modules is actually considered modulo associativity and commutativity. You can, of course, add rewrite rules to your theory corresponding to associativity and commutativity, but in many cases this will be much slower (due to spurious copying) than a dedicated solver for associative-commutative matching. There are a number of theories that can be integrated into the matching algorithm, including:

- Semigroups (A)
- Abelian semigroups (AC)
- Abelian monoids (ACU)
- Idempotent abelian semigroups (ACI)
- Boolean rings (BR)
- Abelian groups (AG)

For a relatively comprehensive review of the literature on equational matching and unification, see [11]. Further research is needed on the asymptotic and in-practice complexity of these matching algorithms to determine which ones are relevant to a practical implementation of equality saturation, but given that the $\mathbb{K}$ Framework implements AC-unification, it is likely that this at least is worth implementing.

4.3 Conditional rewriting

Many optimizations cannot easily be implemented using a term-rewriting system alone; it is often much easier to write optimizations in a *conditional* term-rewriting system, in which rules can have boolean side conditions.

We can improve the expressiveness of our equality saturation algorithm by implementing conditional rewriting. One efficient way to implement this is by eliminating a conditional TRS into an equivalent unconditional TRS using the algorithm described in [8].

5 | Improving correctness

In this chapter we will discuss ways to decrease the chance of user error when using an equality saturation implementation.

5.1 Confluence-checking

Although the term-rewriting systems we accept for equality saturation may not be terminating, it seems likely that they will be confluent. Thus it may be desirable to enforce this condition in the type of `saturate`; it should only accept a confluent term-rewriting system.

How, then, do we define the type of confluent term-rewriting systems? Simple: a pair of a term-rewriting system and a proof certificate of its confluence, in some easy-to-check logic like ZF set theory or type theory. The type of confluent term-rewriting systems will then be an abstract type whose constructor will fail to return if the given proof certificate fails to check (or if the proof certificate is not actually a proof of confluence for the associated TRS). Since we only care about the ability to check a proof certificate and the ability to determine if a given proof certificate is a proof of confluence, we can even be polymorphic in the logic used by defining an appropriate typeclass. The user may want an escape hatch for non-confluent or confluent-but-difficult-to-prove-confluent TRSes, which should be integrated into the logic as a proof of falsehood.

Often, the user will not want to manually prove confluence for their TRS. As it happens, there are a number of pre-existing algorithms for conservatively checking whether a given term-rewriting system is confluent; see [4] for more details. I am not sure if these algorithms emit proof certificates; if they do not, the escape hatch can be used (though obviously in this case it is worth writing extensive tests for the confluence checker to avoid bugs).

5.2 Type-checking

If the user gives us a partial function that assigns sorts to terms with metavariables (which should be the case for any language with lambda syntax and mandatory type annotations) and a subsorting partial order, then we can check the validity of the `Equation`s that make up the `LTRS` given to us by the user.

We do this for each `Equation` by first using the partial function to compute the inferred sorts of the LHS and its metavariables. Then we compute the inferred sorts of the RHS and its metavariables. For each metavariable, we ensure that the sort inferred for the use site in the RHS is a subsort of the sort inferred for the binding site in the LHS. Finally, we ensure that the sort of the RHS is a subsort of the sort of the LHS and vice versa (the subsort relation is reflexive, so this means the two sorts are equivalent).

6 | Example IR formats

6.1 Cartesian closed categories

It is well-known that the internal logic of Cartesian closed categories is the simply-typed lambda calculus. Conal Elliott describes in a 2017 paper [3] a method of converting terms in the simply-typed lambda calculus to algebraic expressions in the signature given by

```
class CCC ((~>) :: * -> * -> *) where
  -- The internal hom of the category
  type (|=>) (~>) :: * -> * -> *
  id      :: a ~> a
  (.)     :: (b ~> c) -> (a ~> b) -> (a ~> c)
  fork    :: (a ~> x) -> (a ~> y) -> (a ~> (x, y))
  exl     :: (a, b) ~> a
  exr     :: (a, b) ~> b
  apply   :: ((a |=> b), a) ~> b
  curry   :: ((a, b) ~> c) -> (a ~> (b |=> c))
  uncurry :: (a ~> (b |=> c)) -> ((a, b) ~> c)
```

Since this representation of the simply-typed lambda calculus has no binders, it can be used with equality saturation, though care has to be taken with the performance heuristic if the code is to be evaluated lazily.

Similar tricks may be possible with other logics internal to categories that are expressible in Haskell, see the chart relating flavors of type theory to flavors of category theory in [7].

6.2 Term-rewriting systems

In unpublished correspondence between Adithya Murali, Alexander Altman, and Remy Goldschmidt (myself), we came up with a method for expressing a term-rewriting system *itself* as an entity that can be optimized using equality saturation.

The datatype equivalent to the type of terms for this system is given as `TermRewritingSystem`:

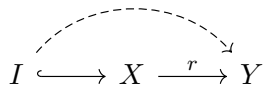
```
type FunctionSymbol = Int
type Label = Text

data Rule = Rule [Label] (OpenTTerm FunctionSymbol) (OpenTTerm FunctionSymbol)

data TermRewritingSystem = TermRewritingSystem (Set Rule)
```

Note that because `TermRewritingSystem` uses the `Set` type in its definition, we need to use AC-matching when implementing this.

The primitive rules used in equality saturation for this language are composition of rules to create new rules (with the lists of labels concatenated) and deletion of an unlabelled rule $r = X \mapsto Y$ if the commutative diagram below is satisfied:



This is a necessary condition for a rule to be deleted; in words, we would describe it as "for every instance I of the pattern term X, there must exist another way to get to Y".

6.3 Relational algebra

Relational algebra, which is a mathematical formalization of a fragment of the languages (like SQL) commonly used for querying relational databases, seems like an ideal language to optimize with equality saturation. It has many equational laws and is already completely referentially transparent.

The same is not true of Datalog, a non-Turing complete fragment of Prolog that is more expressive than relational algebra, but there may be a way to express Datalog queries in a referentially transparent syntax; I suspect it may look like augmenting relational algebra with some kind of fixed-point operator, but I am not sure.

7 | Future Work

7.1 Incremental equality saturation

If there is an algorithm described by a function `f :: A -> B`, we say that `f' :: (A, ΔA) -> B` is the *incrementalized* version of `f` if `ΔA` is a monoid and there exists a monoid action `patch :: ΔA -> A -> A` such that for any `x :: A` and `δ :: ΔA`, `f' (x, δ) = f (patch δ x)`. Ideally, if `x` is already computed and `δ` is "small", the incrementalized version `f' (x, δ)` will also be faster to compute than `f (patch δ x)`, though this connotation is hard to formalize, especially in a lazy language like Haskell.

We can likely usefully incrementalize equality saturation in this sense, as referential transparency ensures that we only need to re-optimize the parts of the PEG that have changed.

This would probably look something like:

1. Compute the difference of the old and new PEGs using a modified version of [6].
2. If there are common subtrees, apply the proofs from the old EPEG to the new EPEG.
3. Exclude the already-explored proof tree branches in the new EPEG.

Alternatively, we may simply be able to reuse the pre-existing EPEG by adding the nodes of the new PEG to optimize to the EPEG, running the maximal sharing algorithm on it, and removing all nodes that are not in the connected component containing the root node of that PEG.

This feature seems especially important for real-world compilers using equality saturation, as it could drastically reduce build times when optimization is enabled and small changes have been made. There are plenty of unexplored questions in this realm; for example, how should the EPEG be quickly and compactly serialized to avoid overhead due to this feature?

7.2 Dependent types

Code written in languages with dependent type systems often ends up containing many theorems and equations encoded in higher-order logic at the type level. These equations could be useful for equality saturation, as they represent (often nontrivial) ways of rewriting a program. Selsam and Moura wrote a 2016 paper [10] on congruence closure in intensional type theory that seems especially relevant to an implementation of equality saturation for a dependently-typed language.

7.3 Homotopy type theory

Homotopy type theory is a constructive foundation for mathematics based on intensional type theory with higher inductive types and the univalence axiom, which is incompatible with the *uniqueness of identity proofs* axiom commonly assumed in dependent type theory.

The main thing distinguishing proofs in homotopy type theory from proofs in dependently typed languages is the fact that there can be nontrivial equalities between type-level equalities. In fact, types in homotopy type theory can usefully be thought of as weak ω-groupoids, which can have arbitrarily tall towers of nontrivial equality. Cubical type theory is an experimental computable implementation of homotopy type theory, so homotopy type theory may have relevance to computer applications.

It would be interesting to explore saturation-based automated theorem proving in the context of these weak ω-groupoids, rather than the 1-groupoids (equivalence relations) we've considered in this paper.

8 | Bibliography

[1] Michael A. Bender et al. "A New Approach to Incremental Cycle Detection and Related Problems". In: *ACM Transactions on Algorithms* 12.2 (Dec. 2015), 14:1–14:22. ISSN: 1549-6325 (cit. on p. 9).
DOI: 10.1145/2756553.

[2] Burton Howard Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782 (cit. on p. 9).
DOI: 10.1145/362686.362692.
URL: https://dl.acm.org/citation.cfm?doid=362686.362692.

[3] Conal Elliott. "Compiling to categories". In: *Proceedings of the ACM on Programming Languages* 1 (Sept. 2017) (cit. on p. 17).
DOI: 10.1145/3110271.
URL: http://conal.net/papers/compiling-to-categories.

[4] Bertram Felgenhauer. "Confluence of Term Rewriting: Theory and Automation". PhD thesis. University of Innsbruck (cit. on p. 16).
URL: http://cl-informatik.uibk.ac.at/workspace/publications/bertram.pdf.

[5] Peter Graf. *Substitution tree indexing*. Research Report MPI-I-94-251. Im Stadtwald, D-66123 Saarbrücken, Germany: Max-Planck-Institut für Informatik, Oct. 1994 (cit. on p. 11).
URL: http://pubman.mpdl.mpg.de/pubman/item/escidoc:1834191:2/component/escidoc:1857890/MPI-I-94-251.pdf.

[6] Eelco Lempsink. "Generic type-safe `diff` and `patch` for families of datatypes". INF/SCR-08-89. MA thesis. Universiteit Utrecht, Aug. 2009 (cit. on p. 19).
URL: http://eelco.lempsink.nl/thesis.pdf.

[7] nLab authors. *relation between type theory and category theory*. http://ncatlab.org/nlab/show/relation%20between%20type%20theory%20and%20category%20theory. Revision 67. May 2018 (cit. on p. 17).

[8] Grigore Roşu. "From Conditional to Unconditional Rewriting". In: *Recent Trends in Algebraic Development Techniques*. Ed. by José Luiz Fiadeiro, Peter D. Mosses, and Fernando Orejas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 218–233. ISBN: 978-3-540-31959-7 (cit. on p. 15).
DOI: 10.1007/978-3-540-31959-7_13.

[9] R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov. "Chapter 26 - Term Indexing". In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 1853–1964. ISBN: 978-0-444-50813-3 (cit. on pp. 9–11).
DOI: 10.1016/B978-044450813-3/50028-X.

[10] Daniel Selsam and Leonardo de Moura. "Congruence Closure in Intensional Type Theory". In: *Automated Reasoning: 8th International Joint Conference on Automated*

Reasoning (IJCAR). Ed. by Nicola Olivetti and Ashish Tiwari. Springer International Publishing, 2016, pp. 99–115. ISBN: 978-3-319-40229-1 (cit. on p. 19).
DOI: 10.1007/978-3-319-40229-1_8.
URL: http://arxiv.org/abs/1701.04391.

[11] Jörg H. Siekmann. "Unification theory". In: *Journal of Symbolic Computation* 7.3 (1989), pp. 207–274. ISSN: 0747-7171 (cit. on p. 14).
DOI: 10.1016/S0747-7171(89)80012-4.

[12] Ross Tate et al. "Equality Saturation: A New Approach to Optimization". In: *ACM SIGPLAN Notices* 44.1 (Jan. 2009), pp. 264–276. ISSN: 0362-1340 (cit. on pp. 2, 3).
DOI: 10.1145/1594834.1480915.