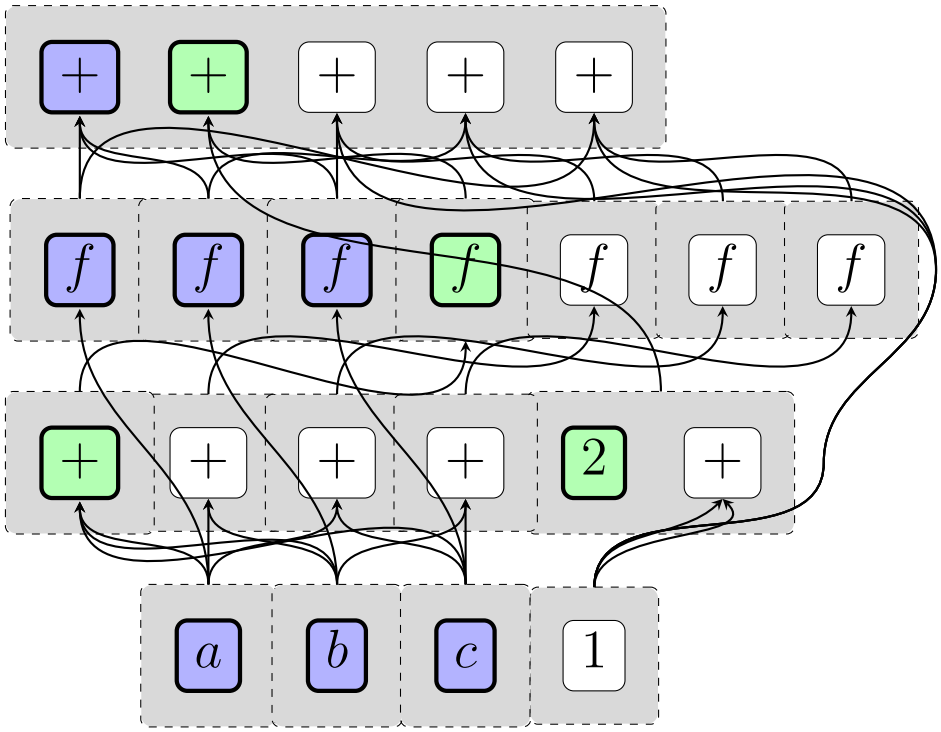
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



E-graphs modulo theories

with applications to associative-commutative operators

Master's thesis in Computer science and engineering

Sofia Brookie

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

E-graphs modulo theories

with applications to associative-commutative operators

Sofia Brookie

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

E-graphs modulo theories
with applications to associative-commutative operators
Sofia Brookie

Supervisor: Nicholas Smallbone, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An E-graph Modulo Theory compactly representing a set of terms involving the associative and commutative function symbol + alongside another function symbol f.

Typeset in LaTeX
Gothenburg, Sweden 2026

E-graphs modulo theories
with applications to associative-commutative operators
Sofia Brookie
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

E-graphs are a data structure for equational theorem proving which provide the basis for *equality saturation*, a method to automatically derive equational consequences from a starting set of terms, equations, and identities. E-graphs have generated a lot of interest since the publication of the Egg paper by Willsey et al. [23], which describes a fast implementation of E-graphs and its application to a variety of problems.

However, E-graphs struggle with a combinatorial explosion when reasoning about certain theories. A notorious case is that of AC theories, those containing associative and commutative operators: the space complexity of saturation is exponential (more precisely, $O(3^n)$) in the size of the input in the worst case.

Can equality saturation for AC theories be done in a more efficient way? There have been several proposals for extending E-graphs to *E-graphs modulo theories* (EMTs) [26, 14, 17]. These proposals suggest integrating a *theory-specific* reasoning method for AC together with the general mechanisms of E-graphs and equality saturation in order to efficiently deal with AC theories. The hope is that specific methods for AC theories can avoid the inefficiencies in using the general equality saturation mechanism for the AC identities.

However, there is a limitation in previous proposals for EMTs for AC, which cause the resulting notion of EMT to be unable to determine straightforward consequences of the AC identities. This thesis develops a notion of EMT that addresses this issue and shows that it *is* possible to reason with AC theories and a variety of closely related theories in a way that avoids the worst inefficiencies of plain E-graphs. It also provides a formal definition of EMTs that we hope can provide the basis for future work on integrating other theories in a similar manner. Finally, we discuss an implementation of EMTs and benchmark it against pure E-graph reasoning on a variety of equational reasoning tasks (5.3), showing significant space savings compared to ordinary E-graphs.

Central to our notion of a 'good' theory for an EMT is the necessity of a solver for the word problem over the theory. This gives us a lower bound on EMTs for AC: the word problem for AC requires in the worst case exponential space and doubly-exponential time. This is an improvement over the space upper bound for ordinary E-graphs, but highlights that concrete efficiency gains derive from the potential for typical problems to be solvable efficiently. Our solver for AC and polynomials builds upon Buchberger's algorithm, on which significant work has been done to achieve good common-case performance. Theories similar to AC, such as those with identities for units, negation, or scalar multiplication, have word problems

solvable via Buchberger-like algorithms, and in certain cases, AC-like theories can be solved significantly more than AC by itself. Thus our work provides a foundation for future generalization to a this large family of AC-like theories.

Keywords: E-graphs, equality saturation, AC, associativity, commutativity, Buchberger's algorithm, rewriting.

Acknowledgements

Thank you to my supervisor, Dr. Nick Smallbone, who first noticed the flaw that led to the focus in this thesis on the word problem/AC-closure rather than matching. Thank you to Dr. Hanna Svensson for proofreading. Thank you to Tarik Rosin for early access to their work on AC-saturation, without which this thesis would still likely have the incorrect asymptotics for E-graph size under AC-saturation. Thank you to Milo the cat for occasionally allowing me to use his chair in front of my computer.

Sofia Brookie, Palmerston North, 2026-06-28

Contents

List of Figures

List of Tables

1

Introduction to E-graphs

1.1 Equational theorem proving

In equational theorem proving, the problem we are interested in solving is how to take a set of equations and find the set of consequences of those equations. There two different cases to consider:

1. We know the goal equation, so we want to either *validate* that it follows from the input equations or determine that it doesn't follow
2. We do not know the goal equation, and we want to explore some interesting equations that follow from the input equations

E-graphs can be used for both, but they particular shine in the second case, as they are not fundamentally 'goal-directed'.

Let us consider an example.

$$f(x, y) = f(y, x) \tag{1.1}$$
$$f(1, 1) = 2 \tag{1.2}$$
$$f(2, 1) = 3 \tag{1.3}$$

Here, the first equation is understood to hold for any values of x and y. We will call such an equation an *identity*: really, it stands for the universally quantified formula $\forall xy.\ f(x, y) = f(y, x)$. We will then reserve the term 'equation' for *ground* equations: those equations without such quantifiers.

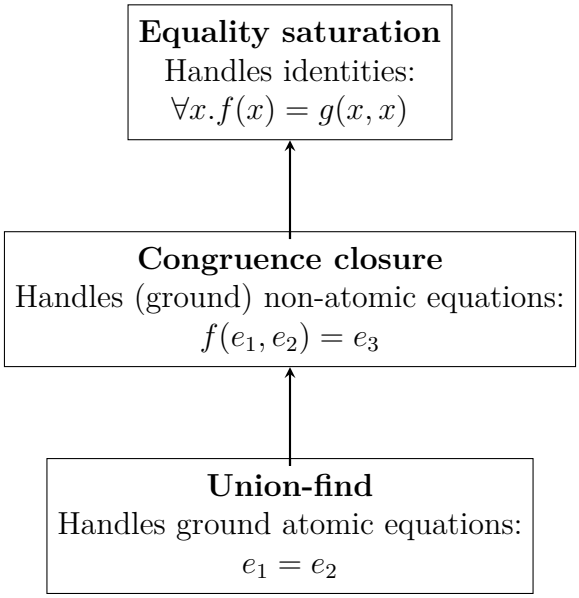
Now, here is one consequence of the above equations:

$$\begin{aligned} & f(1, f(1, 1)) & \\ = {} & f(f(1, 1), 1) & \text{by equation 1.1} \\ = {} & f(2, 1) & \text{by equation 1.2} \\ = {} & 3 & \text{by equation 1.3} \end{aligned}$$

E-graphs are a data structure on which we can run *equality saturation* to discover the above equational consequence automatically, alongside other consequences.

Applications of equality saturation include optimization in compilers, decompilation, program synthesis, and hardware synthesis [23, 7, 24]

1.2 Union-find, congruence closure, and equality saturation



Describing E-graphs can be done in 3 layers. First, there is the union-find structure, which is a simple solver for the theory of atomic equations: given a set of *atoms* $e_1, e_2, \ldots$, we can union two of them to set them equal, and we can find a representative of any particular atom. The representative is unique, so we can simply compare representatives for equality to determine if the atoms they represent are equivalent.

Union-find alone cannot handle equations between non-atomic terms, like $h(a) = f(g(c))$. In particular, terms cannot be treated as atomic, since there is the rule of *congruence*: if $s_i = t_i$, then we must conclude $f(s_1, \ldots) = f(t_1, \ldots)$.

To handle non-atomic terms, we build a congruence-closure structure on top of a union-find structure. The congruence-closure structure works by breaking terms down into *layers*. For instance, to represent $f(g(e_1), e_2)$ we will assign a new constant e_3 for $g(e_1)$, and then assign a constant e_4 for $f(e_3, e_2)$. This way, all congruence checks only need to apply to 'flat' terms: if $e_i = d_i$ then we conclude $f(e_1, \ldots) = f(d_1, \ldots)$.

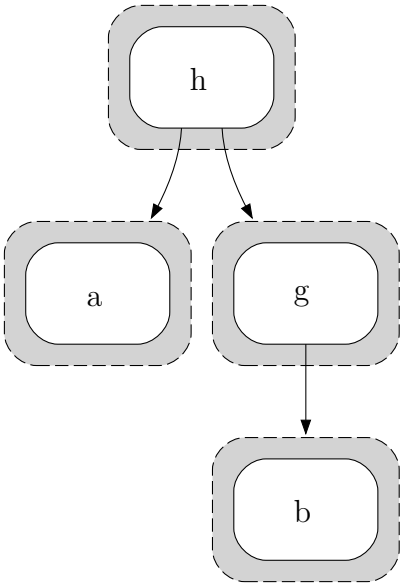
This motivates the E-graph data structure, and the congruence closure algorithm that restores the congruence invariant. Computing the congruence closure is also known as 'rebuilding' the E-graph.

1.3 E-graphs for congruence-closure

As an example, we will compute the congruence closure of the following set of equations:

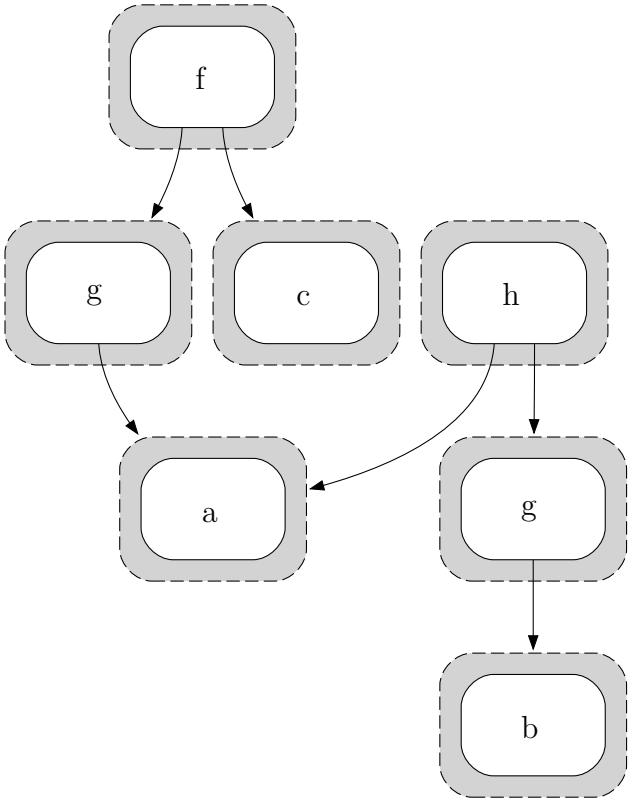
$$f(g(a), c) = h(a, g(b)) \tag{1.4}$$
$$h(g(c), b) = h(a, g(c)) \tag{1.5}$$
$$g(b) = g(c) \tag{1.6}$$
$$f(g(a), c) = g(b) \tag{1.7}$$

As a first step, we represent the term $h(a, g(b))$ from equation 1.4:

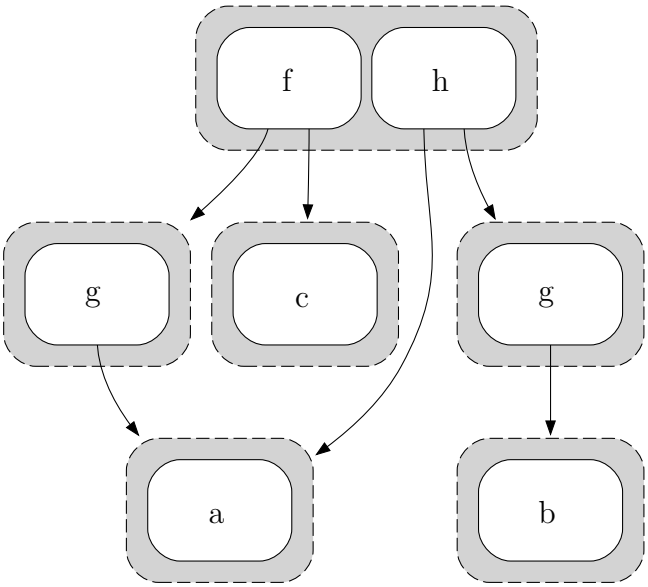


Notice how we have flattened the term: the three levels of nesting become edges or pointers between the three layers of our graph. In particular, we think of the edges from h as being numbered: the left-most edge corresponds to the first argument $h(a, -)$, while the rightmost is the second argument $h(-, g(b))$.

Then we will add the other term $f(g(a), c)$ from equation 1.4:



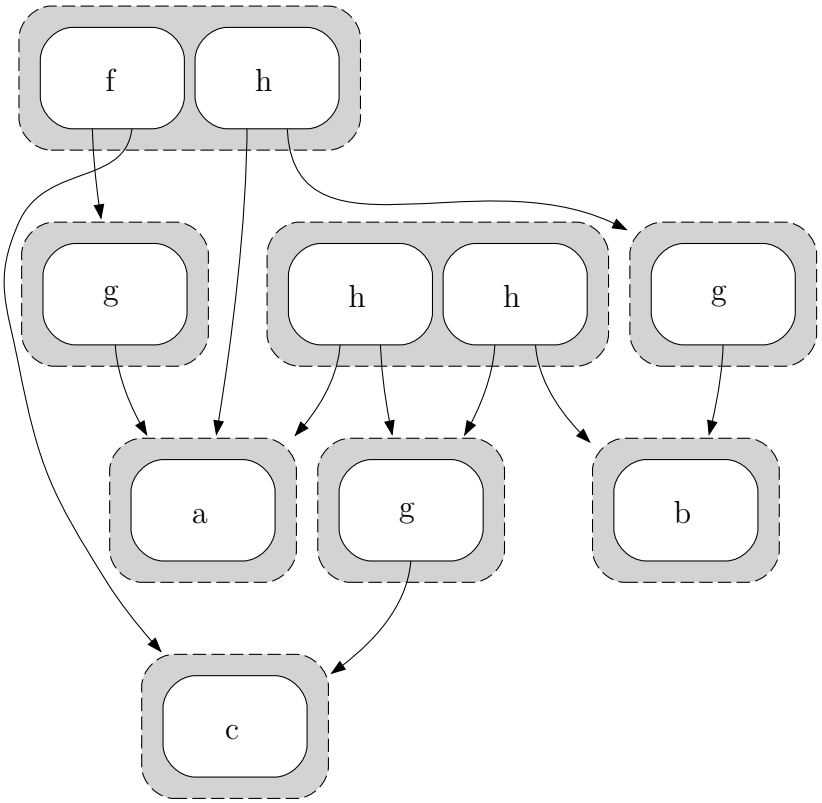
Now, as a final step, we need to *union* the two top nodes of both sides of the equation:



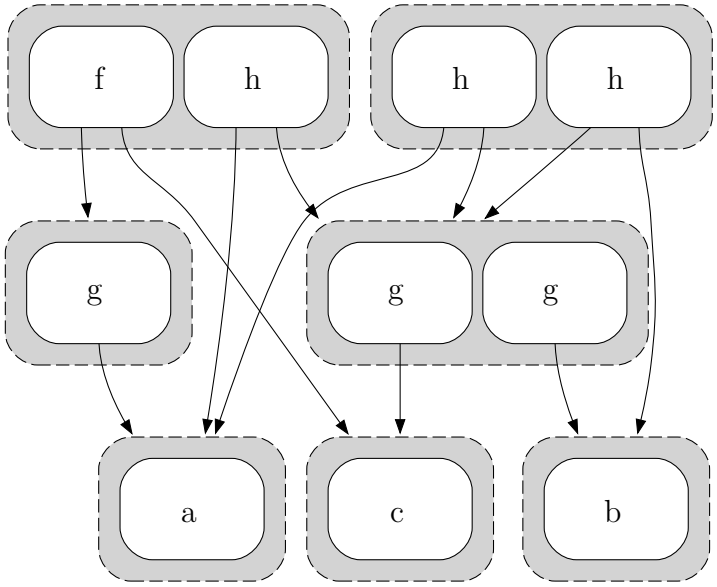
We call the outer nodes *E-classes* (equivalence classes), while the inner nodes are *E-nodes*. E-classes can contain multiple E-nodes, and two E-nodes are in the same E-class iff they represent equivalent (sets of) terms. The arrows point from E-nodes to E-classes. So, for instance, there are two E-nodes each with the symbol g, each as

the only item in their respective E-classes. The only non-trivial E-class so far is the top E-class, with two E-nodes labelled with f and h.

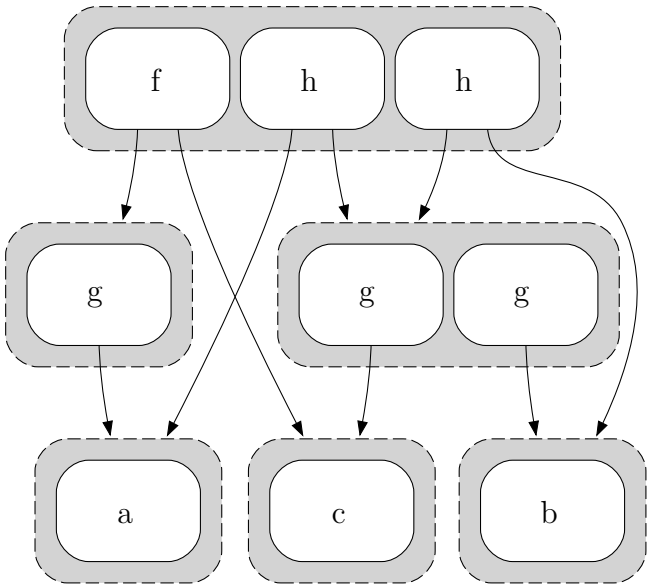
Next we move on to equation 1.5 ($h(g(c), b) = h(a, g(c))$). This time, notice that the subterm $g(c)$ is used twice, and we *share* this in the E-graph, indicated by two arrows pointing to the E-class for $g(c)$:



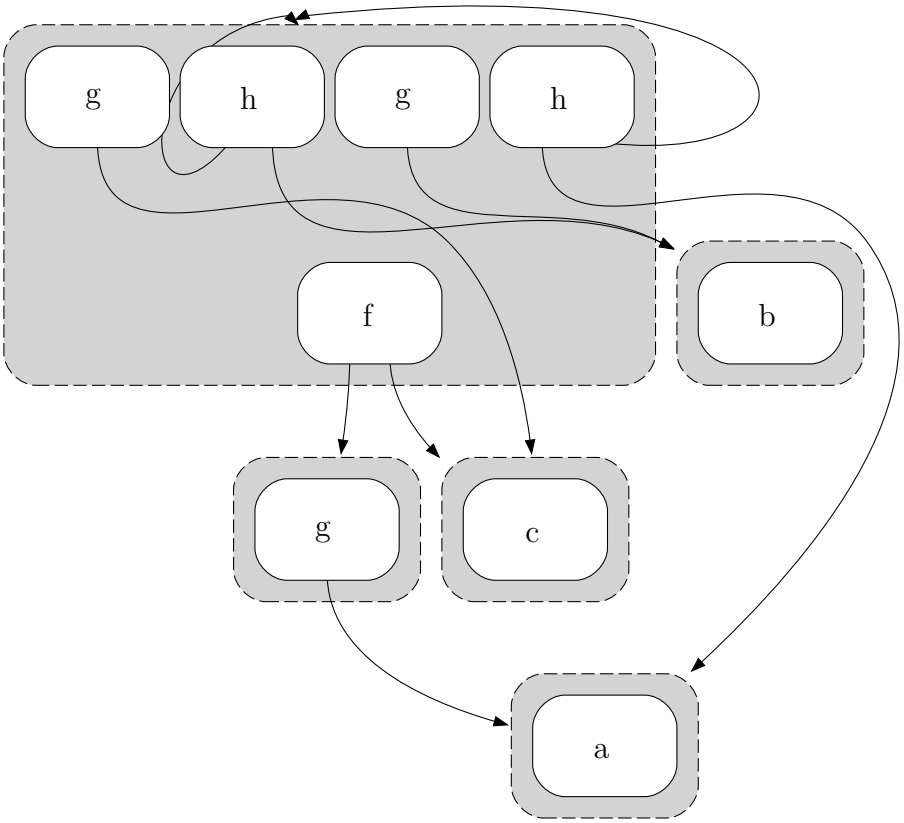
Now we handle equation 1.6 ($g(b) = g(c)$), which does not add any new E-nodes in the graph. Instead, it merges the separate E-classes representing $g(b)$ and $g(c)$ into a single E-node:



Notice that we have two instances of an h with the same arguments: $h(a, g(b))$ and $h(a, g(c))$ have become equal by congruence, but they are located in two different E-classes. So, after unioning, we need to *propagate* by continuing to union E-classes that have become equal. This gives us the simpler E-graph:



Finally we handle equation 1.7 ($f(g(a), c) = g(b)$). This time, it will create a *cycle* in the E-graph: an E-node pointing to its own E-class.



1.3.1 The congruence closure algorithm and cyclic E-graphs

To compute congruence closure, we can use a fix-point algorithm: find some E-classes that need to be merged, merge them, and repeat until there is nothing left to do. However, showing that this terminates is slightly non-trivial, as E-graphs can contain cycles.

A simple equation that generates a cycle is $g(d) = d$, which gives rise to the infinite set of equations $g^n(d) = d$ by congruence.

In general, we might have cycles going through multiple intermediary nodes:

$$f(\ldots g(\ldots e_1 \ldots) \ldots) = e_1$$

Cycles become very important in the analysis of how theories fare with E-graphs under equality saturation (section 6.2).

Handling congruence closure in E-graphs with cycles is automatic if we use the above fix-point algorithm. There is a finite upper bound on propagation: at most, we could discover that every E-class is equal to every other. Every step moves closer to that bound by merging some E-classes. As there is a bound on how many merges are possible, this algorithm clearly terminates.

1.4 Equality saturation

Equality saturation is a procedure that builds on the congruence-closure algorithm for E-graphs to handle *identities* rather than just *equations*.

Suppose that, instead of equation 1.6 ($h(g(c), b) = h(a, g(c))$) in the example from above (section 1.3), we had the identity $\forall x.\ h(a, x) = h(x, b)$. To do equality saturation with this identity, we would *match* the left-hand-side $h(a, x)$ against our E-graph, which would find the match $h(a, g(b))$, corresponding to the substitution $x = g(b)$. Then, we apply the same substitution to the right hand side, thus finding the *equations* $h(a, g(b)) = h(g(b), b)$. Inserting this equation into our E-graph and using congruence closure would then yield the same E-graph as above.

Equalities are symmetric, so we also need to match the right-hand-side and add the substituted left-hand-side in general equality saturation.

So, in general, to handle identities, we use the following procedure:

1. Add every ground equation to the E-graph
2. Equality saturation outer loop:
   (a) (E-match) Match any side of an identity to a ground term represented by the E-graph
   (b) (Insert) the other side: using the substitution from the previous step, construct the term resulting from applying the substitution to the other side of the identity. If it is not already represented by the E-graph, add it to the E-graph.
   (c) (Merge) the E-classes representing the two sides from above
   (d) Recompute congruence closure (section 1.3.1)
   (e) Repeat

If the fixed point is reached, we call the graph *saturated*.

In practical applications such as compilers, we often have a final step to *extract* some minimal term equal to the starting term. Extraction is covered in section 4.2.

1.4.1 Aside: variations on equality saturation

We can do several matches/insertions/merges in between rebuilding; it is sufficient that rebuilding is *eventually* run after changes to the E-graph [23].

There are plenty of extensions to the outer loop that include things such as conditional reasoning (along the lines of: if $a = b$, then assert that $c = d$ by merging classes) or E-class analyses [23].

1.4.2 Equality saturation: limitations

When both substituted terms from E-matching an identity are already represented by the graph, we call the merge a *proper merge*. Unfortunately, proper merges are not enough. If we consider the associative identity $\forall xyz.\ x + (y + z) = (x + y) + z$, and we begin with the two terms $w + (x + (y + z))$ and $((w + x) + y) + z$, we would like to prove that these terms are equal. However, the attempt to prove even the first step, that $w + (x + (y + z)) = (w + x) + (y + z)$, founders. Our representation includes no equivalence class to represent $(w + x) + (y + z)$, and so there are no equivalence classes to merge.

The 'solution' is to include the possibility of *inserting* new classes during the procedure. We call a mix of insertion and a merge an *improper merge*.

Insertion has one key problem: we now have no guarantees the process of merging will terminate. With only proper merges, there is a finite set of input terms and thus a finite bound on the equivalence relation we maintain. Thus, eventually we will get to a point where no merge changes the equivalence relation, and we have reached saturation. Introducing new classes for inserted terms now means that saturation is no longer guaranteed to be reached at any point, and worse, the strategy for insertion and merging now can affect whether or not saturation is reached.

With the possibility of non-termination, we might wish to at least get a *semi-decision procedure*: the procedure takes in a specific equality and must prove the equality if it is provable, but can loop indefinitely when it isn't. For this, we must ensure that the insertion and merging processes are sufficiently *fair* (in the sense of 'fair' as used in concurrent programming): we must have some strategy to ensure every possible term that could be inserted *does* get inserted eventually, or we might get stuck inserting other terms infinitely while nonetheless not making progress on the goal.

In some sense, this is to be expected, as even for just equational theorem proving, there are theories which are known to be undecidable. However, even when the process of E-matching, inserting and merging does not terminate, we can consider the infinite fixed-point of this process as the 'semantics' of our E-graph [18], and this will be what we will preserve in the shift to EMTs.

2

E-graphs modulo AC

When doing using E-graphs, one common pair of identities we want to work with is associativity

$$\forall xyz.\ f(x, f(y, z)) = f(f(x, y), z) \tag{2.1}$$

and commutativity

$$\forall xy.\ f(x, y) = f(y, x) \tag{2.2}$$

We refer to the combination of these identities as AC. Developing a method for working modulo AC efficiently is a central goal of this thesis.

A common AC operator is $+$, and we will use this as our main example of an AC operator.

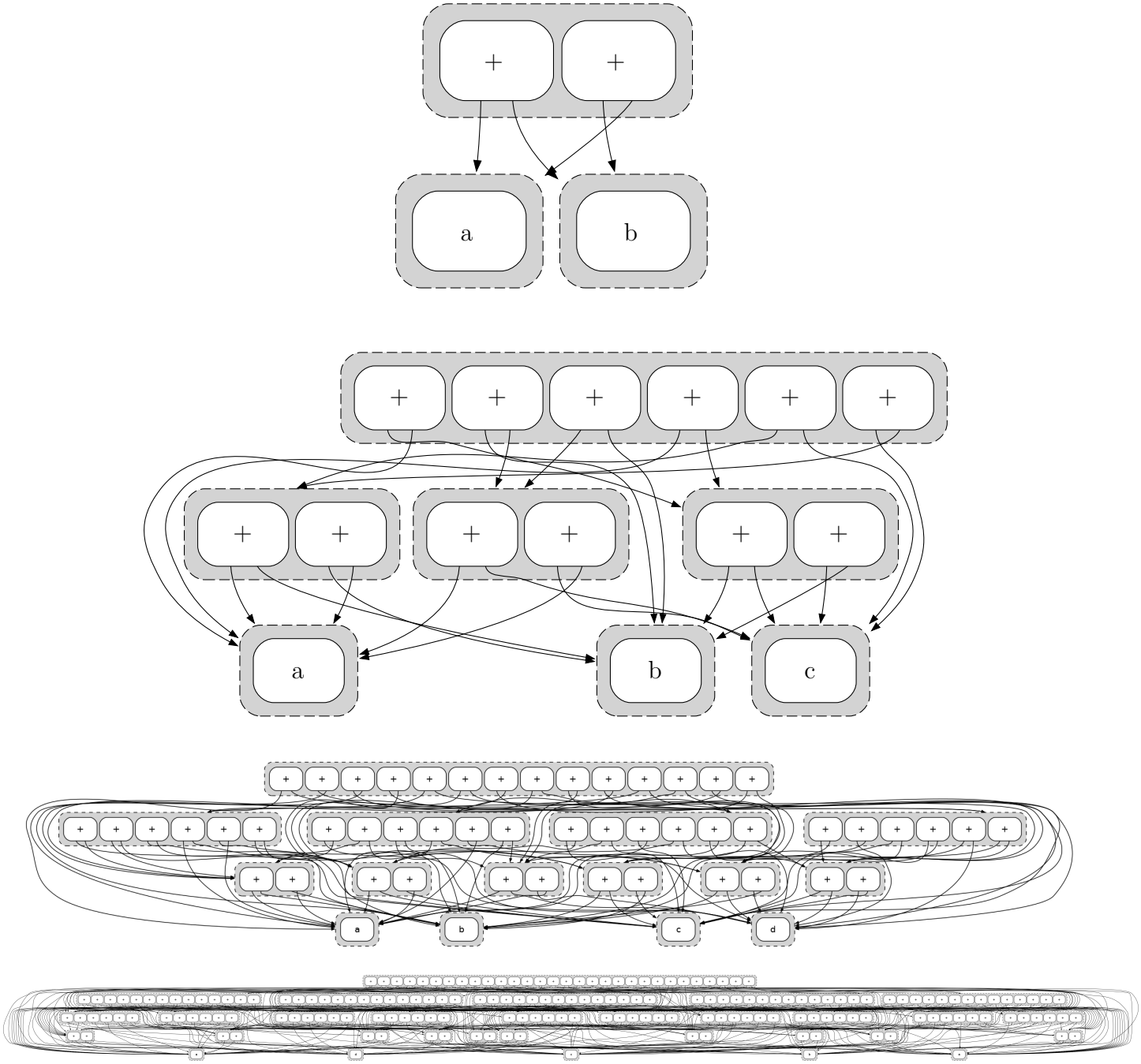
2.1 AC and equality saturation

Unfortunately, equality saturation handles the AC identities rather poorly. Consider as a simple case the terms $t_n = a_1 + a_2 + \ldots + a_n$, where we add together n constants.

There are many ways to parenthesize this term when n is large. For instance, for $n = 5$, we have $a_1 + ((a_2 + (a_3 + a_4)) + a_5)$ and $(a_1 + a_2) + (a_3 + (a_4 + a_5))$ and many other parenthesizations, all of which are equal up to A. The exact number of such parenthesizations is given by the n-th Catalan number C_n, which is $O(\frac{4^n}{n^{3/2}})$.

Adding commutativity means we can reorder the n constants in any of $n!$ ways. For instance, $a_1 + (a_2 + a_3)$ is equal to $(a_3 + a_1) + a_2$ under AC.

Running under the AC-identities to saturation, the E-graph will need to represent that all of these ways of rearranging the term are equal. E-graphs are able to share some of the subterms that are required to represent all these equalities, but running to saturation still requires exponentially-sized E-graphs:



Theorem 1. *Saturating the term $a_1 \dots a_n$ under AC yields an E-graph with $2^n - 1$ E-classes and $3^n - 2^{n+1} + n + 1$ E-nodes [16].*

Proof. Repeated applications of the AC-identities will create one E-class representing the sum of every non-empty subset of the set $\{a_1, \dots, a_n\}$.

Each such class representing the sum of a subset A_i of $\{a_1, \dots, a_n\}$ will have one E-node $+(e_j, e_k)$ for every pair of E-classes e_j and e_k such that the E-classes e_i and e_j represent a partition of the terms in the sum: $A_i = A_j \cup A_k$.

So, for an E-class where $|A_i| = m$, we have $2^m - 2$ E-nodes: 2^m subsets of A_i for the

left partition A_j, minus the total and empty partitions, since there is no E-class for the empty sum. For the special case $|A_i| = 1$, we have 1 E-node to represent the single constant in A_i.

The total amount of E-nodes is therefore

$$\begin{aligned}
&\sum_{m=2}^{n} \binom{n}{m}(2^m - 2) + n \\
&= (\sum_{m=0}^{n} \binom{n}{m}(2^m - 2) - \binom{n}{1}(2^1 - 2)) - \binom{n}{0}(2^0 - 2) + n \\
&= \sum_{m=0}^{n} \binom{n}{m}2^m - 2\sum_{m=1}^{n} \binom{n}{m} + 1 + n \\
&= \sum_{m=0}^{n} \binom{n}{m}2^m 1^{n-m} - 2\sum_{m=0}^{n} \binom{n}{m}1^m 1^{n-m} + 1 + n \\
&= (2+1)^n - 2(1+1)^n + 1 + n \\
&= 3^n - 2^{n+1} + n + 1
\end{aligned}$$

□

We can summarize these combinatorial results as there being $O(3^n)$ E-nodes and $O(2^n)$ E-classes at saturation.

We refer to this approach to handling AC by using the ordinary mechanisms of equality saturation as *AC-saturation*.

2.2 Handling AC with EMTs: a first example

The poor handling of AC by plain E-graphs with equality saturation motivates developing a notion of *E-graphs modulo theories*. In this case, we want to work 'modulo AC' in some way.

We will use an example problem: besides the AC operator $+$, we introduce a function symbol f and the identity $\forall xy.\ f(x+y)+1 = f(x)+f(y)$. We have integer constants satisfying the identities of arithmetic, in particular the equation $1 + 1 = 2$.

We begin with the term $f(a) + (f(b) + f(c))$. We would like to discover that this input term is equal to $2 + f(a + (b + c))$, using the reasoning

$$\begin{aligned}
&f(a) + (f(b) + f(c)) \\
&= f(a) + (f(b + c) + 1) \\
&= (f(a) + f(b + c)) + 1 \\
&= (f(a + (b + c)) + 1) + 1 \\
&= f(a + (b + c)) + (1 + 1) \\
&= f(a + (b + c)) + 2 \\
&= 2 + f(a + (b + c))
\end{aligned}$$

We will present a pictorial overview of how we might solve the above problem using equality saturation on EMTs rather than E-graphs.

The natural first step is to simply not add any different representations of terms that are equal up to AC; thus, nodes in the E-graph for AC operators will be variadic and we will consider their children unordered. This is the key idea from prior work [26, 14, 17], and is one attempt to define what 'working up to AC' means.

Figure 2.1 begins with the variadic AC-term $f(a) + f(b) + f(c)$. We omit the labels of equivalence classes.

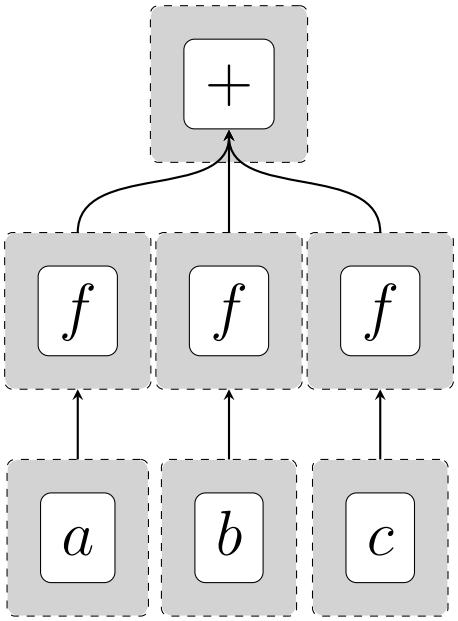


Figure 2.1: EMT of the term $f(a) + f(b) + f(c)$.

Working up to AC in this way, we don't need to pick an association $f(a)+(f(b)+f(c))$ or $(f(a) + f(b)) + f(c)$ for the input, and all associations and all reorderings are considered to be represented by the top node in figure 2.1.

Now, in order to *match* the left-hand side of $f(x) + f(y) = f(x + y) + 1$, we need to do *matching modulo theories*, which will tell us of six matches: $(x, y) \in \{(a, b), (a, c), (b, c), (b, a), (c, a), (c, b)\}$. Then we will add the six substituted forms of $f(x + y) + 1$. Here, we will want to *insert modulo theories*, in a way that keeps the E-graph from representing multiple AC-equivalent terms. The arrows in Figure 2.2 have become crowded, but top E-class now represents the original term, and $f(a + b) + f(c) + 1$ and its variants with a, b, c permuted.

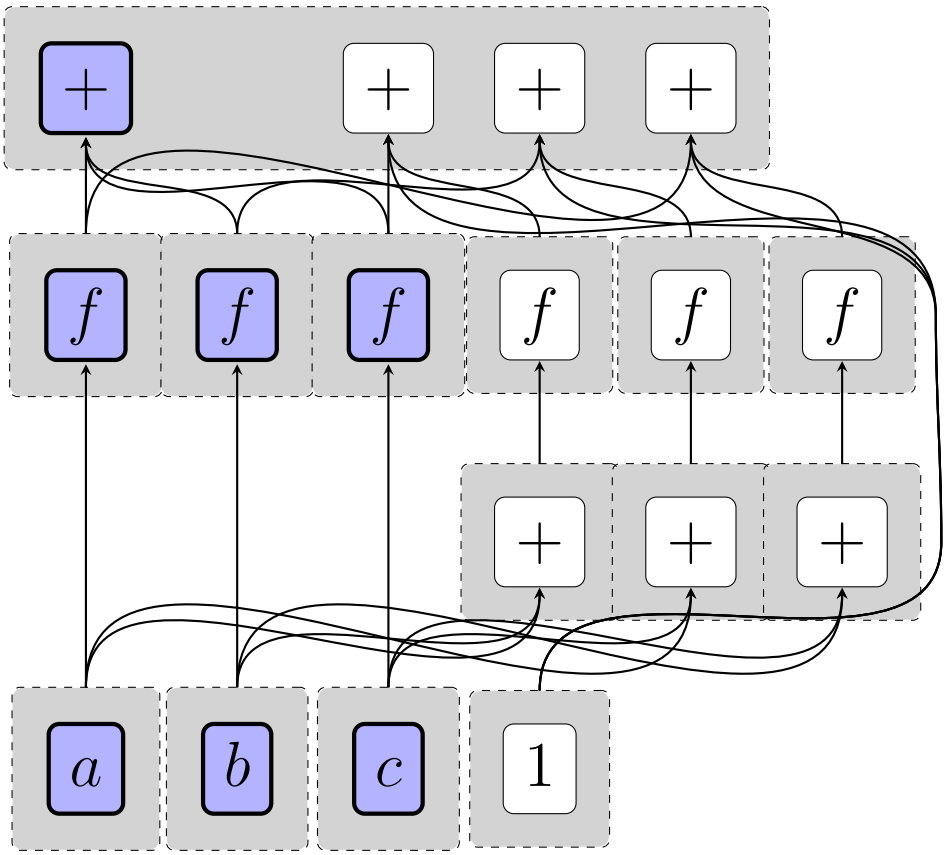


Figure 2.2: EMT of the term $f(a) + f(b) + f(c)$ after some steps. The initial term is highlighted.

Now, we once again try to match $f(x) + f(y)$. The only new matches will be $(x, y) = (a + b, c)$ and its permutations, and, modulo AC, the substituted-for term $f(x + y) + 1$ will be the same for all of these. So, we just need to add a single node.

Our last step is to notice that the term $f(a+b+c)+1+1$ has, up to AC, the implicit subterm $1 + 1$, which we can reduce to 2. Having reduced the implicit subterm, we no longer need the 3-part term $f(a + b + c) + 1 + 1$, and can entirely replace it with $f(a + b + c) + 2$.

The final EMT (Figure 2.3) is saturated, and has 'discovered' our desired equality, while remaining relatively compact.

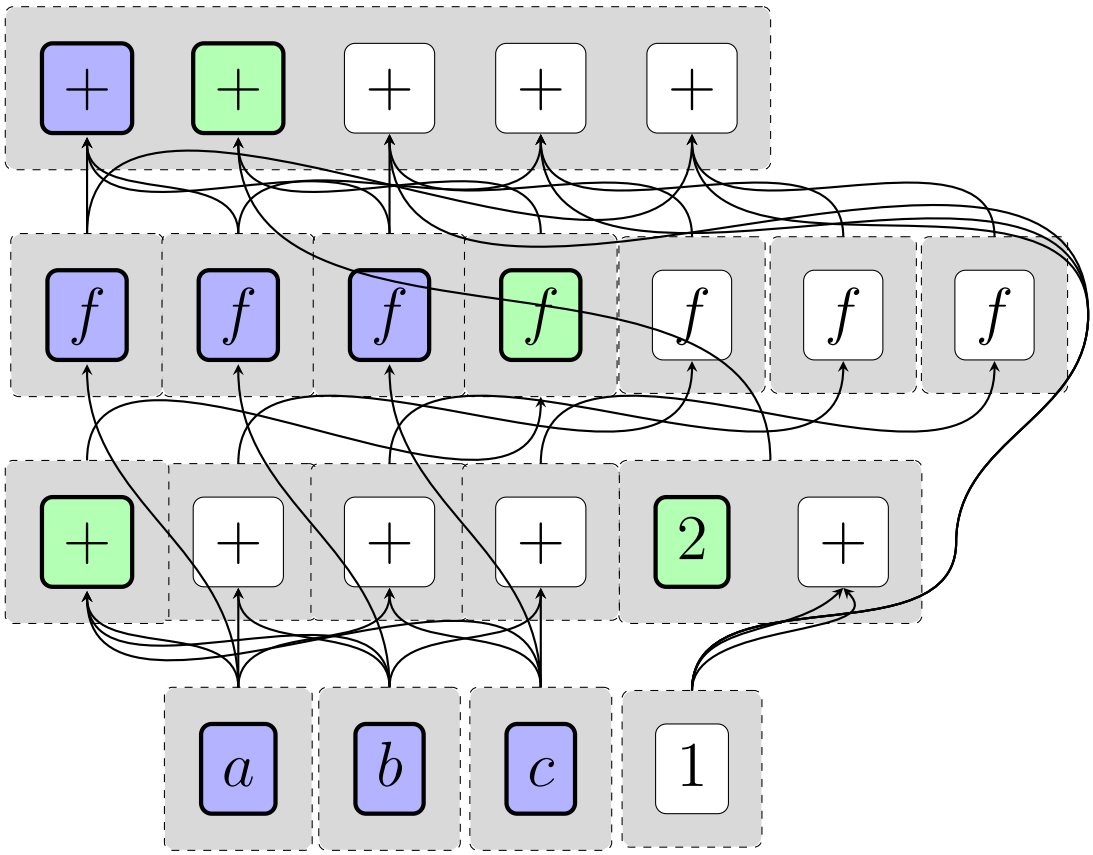


Figure 2.3: EMT of the term $f(a) + f(b) + f(c)$ once saturated, highlighting the initial term and the desired final term.

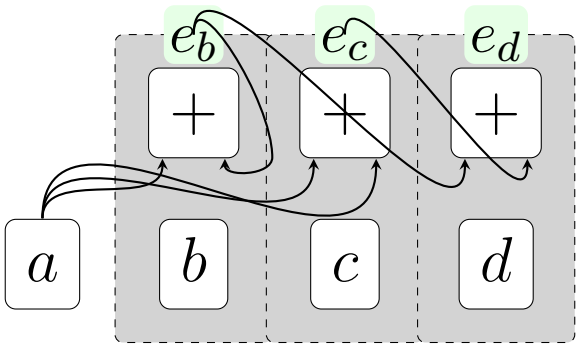
2.3 Challenges for EMT(AC)

We have seen the promising aspects of EMTs for AC, but there is a significant gap in the description given so far, and we need something extra. Consider the equations

$$a + b = b \tag{2.3}$$
$$a + a = c \tag{2.4}$$
$$c + b = d \tag{2.5}$$

taken modulo associativity of $+$.

We can represent them with an E-graph as follows:



Consider the equation $b = d$, which follows from the simple deduction

$$b = a + b = a + (a + b) = (a + a) + b = c + b = d$$

However, e_b and e_d are two distinct E-classes in the E-graph! Why?

The central step in the deduction relies on two different associations of the term $a + a + b$. The association $(a + a) + b$ is correctly assigned to class e_d, while the association $a + (a + b)$ is correctly assigned to the class e_b. However, given that we want to work modulo theories, we need to discover the fact that $e_b = e_d$, or we are throwing out some of the consequences derivable from the theory.

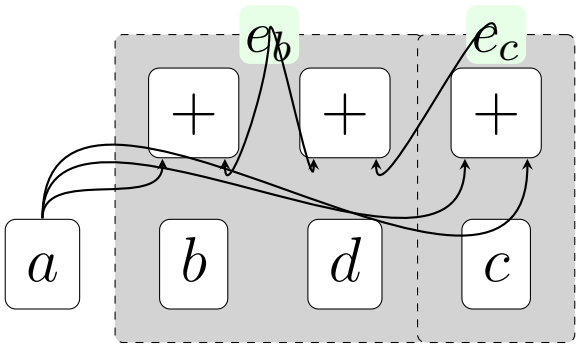
2.4 Filling the gap

To fill the gap, we need to reason about when such an 'overlapping' term could occur, while not generating all AC-subterms like equality saturation would.

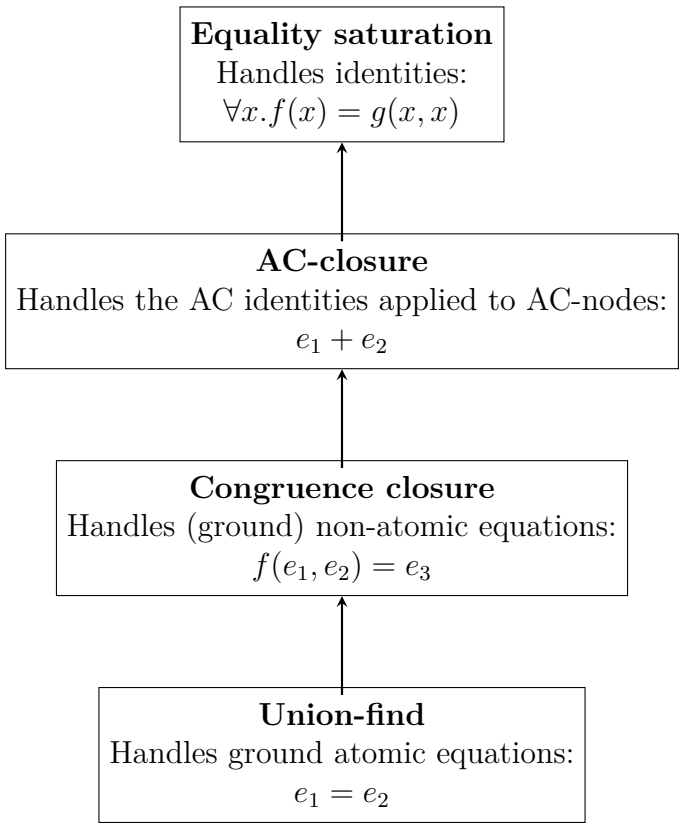
Our criteria is this: if the EMT contains two terms $t_1 = r + s$ and $t_2 = u + s$, with a non-trivial *overlap* s, then we need to insert the equation $t_1 + u = t_2 + r$.

This new equation is a consequence of the AC identities, since $t_1 + u = (r + s) + u = (u + s) + r = t_2 + r$.

With our above example, we see that e_b contains the E-node $+(a, e_b)$ and e_c contains the E-node $+(a, a)$. Consequently, we insert the equation $e_b + a = e_c + e_b$. Since the right-hand-side is already assigned to e_d and the left-hand-side to e_b, this equation simply asks us to equate e_b and e_d, exactly as desired.



We refer to the procedure of adding these new equations as *AC-closure*. We can think of it as fitting into the standard parts of E-graphs like this:



AC-closure occupies a middle ground between congruence closure and equality saturation: it is more general than congruence closure and less general than equality saturation, as it handles the AC identities but no others, and it has an intermediate time complexity between polynomial-time congruence closure and equality saturation, which is generally not even guaranteed to terminate.

2.5 Cycles and AC-saturation

AC-saturation (handling the AC identities with equality saturation) will in general not terminate on terms with certain kinds of cycles while AC-closure will. Consider the equation

$$a = a + b \tag{2.6}$$

This is a generic example of a cycle through the operator $+$, in the sense that if there is some E-class e_a that is equal to a sum that includes itself $e_a = \ldots + (\ldots + e_a + \ldots) + \ldots$, it can be put into this form by applications of AC.

Since $a = a+b = (a+b)+b$, equality saturation will generate the equation $(a+b)+b = a+(b+b)$ to be added to the E-graph. Then, $a+(b+b) = (a+b)+(b+b) = a+(b+(b+b))$, so the term $b + (b + b)$ will be added to the E-graph, etc. Therefore, AC-saturation will add an infinite number of terms.

AC-closure does not have this problem: there is an overlap between a and $a + b$ on a, which generates the new equation $a = a$, which is trivial.

Of course, this example may be a little artificial as, in typical domains of interest, $a + b = a$ means that $b = 0$, so that $b + b = b$[1]. Once the E-graph has added the equation $b + b = b$, this will collapse any further term built only of bs down to just b. Or, in other words, any new equation $a + (b + \ldots (b + b))$ will already be implicitly represented in the E-graph.

Still, this illustrates one more advantage of AC-closure over AC-saturation, and is an example that illustrates some of the subtlety of cycles in E-graphs.

2.6 Prospects for EMT(AC)

EMT(AC) avoids the need for equality saturation on the A and C identities by integrating these rules into the underlying E-graph. However, this has a cost: congruence closure normally is polynomial time[2] while the AC-closure step outlined above is at worst ESpace[3] as will be discussed in section 6.3.2.

Moreover, while congruence closure only shrinks the graph by potentially discovering E-classes that can be merged, AC-closure can grow it, so one might want to run AC-closure only intermittently while running congruence closure much more often. This is then not that different in spirit to existing approaches to handling AC blowup in equality saturation by 'deprioritizing' the AC identities[4]

However, there are two key advantages:

- In many practical cases, the AC-closure is small, while ordinary equality saturation will have to run to completion to be 'sure' that nothing is missed.

[1] Or, for the second-most famous AC operator $\times$, if $a \times b = a$, $b = 1$

[2] indeed, at best slightly superlinear

[3] space proportional to $2^{O(n)}$, where n is the size of the E-graph

[4] For example, see the code for `BackoffScheduler` in Egg [22]

- AC-closure is much better at finding 'interesting' nodes to add to the E-graph, in the sense that the added nodes are part of some nontrivial equality $t_1 + u = t_2 + r$. Equality saturation on AC will generate many 'trivial' identities alongside such potentially useful ones.

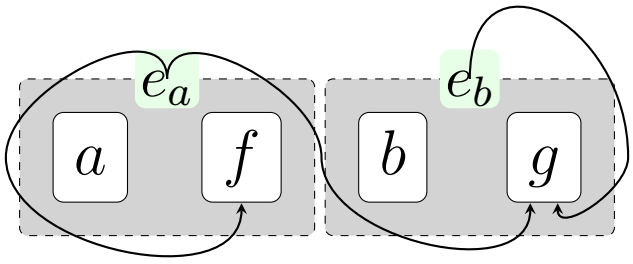
3

E-graphs, EMTs, and rewriting

In order to more precisely specify EMTs for AC, we introduce some ideas from term rewriting theory. In particular, both E-graphs and EMTs can be seen as representations of certain systems of rewrites, and the algorithms and correctness proofs can be described in terms of these rewrite systems.

3.1 E-graphs present a rewriting system

Consider the E-graph



We can read off a set of equations from it:

$$\begin{aligned} a &= e_a \\ f(e_a) &= e_a \\ b &= e_b \\ g(e_a, e_b) &= e_b \end{aligned}$$

The equations can be constructed as follows: for every E-node in E-class e_j, we create an equation $f(e_i, \ldots) = e_j$ where f is the function symbol of the E-node and $e_i, \ldots$ are the children of the E-node, i.e., the E-classes with edges going into the E-node in question. We recall that the edges in the E-graph are thought of as numbered, or in other words, as pointing from an E-class to a specific numbered *argument slot* on the E-node.

Every equation in this set will be of the form $f(e_1, \ldots,) = e_j$, with a function symbol applied to some (possibly 0) E-classes on the left and an E-class on the right.

Given this resulting set of equations, we can determine if any two terms at all are equal according to these equations and according to the underlying E-graph: simply reduce the terms by treating these equations as a *rewrite system*, read from left-to-right, and see if the normal forms are the same. This is the perspective of Abstract Congruence Closure [3]: E-graphs present a rewriting system for terms over an *extended signature*. The signature is extended in the sense that we don't just have the symbols f, a, g, etc, but also $e_a, e_b, \ldots$. In general, we introduce a new E-class e_i whenever we need to represent a subterm.

Then the congruence closure algorithm, aka E-graph rebuilding or propagation, is a matter of restoring *confluence* to this rewrite system. Confluence is the property that any maximal sequence of rewrites end up with the same result. For instance, consider our example on page 6, where we discover that $h(a, g(b))$ and $h(a, g(c))$ are equal by congruence. Two rules we can read off the E-graph are:

$$h(e_a, e_2) \to e_3$$
$$h(e_a, e_2) \to e_4$$

where we assign the name e_a to the E-class containing the E-node a, e_2 to the E-class representing both $g(b)$ and $g(c)$, and assign e_3 and e_4 to the top-left and top-right E-classes.

$h(e_a, e_2)$ can rewrite to both e_3 and e_4. This would violate confluence, unless we have the rewrite $e_3 \to e_4$ or $e_4 \to e_3$. Suppose we add the rule $e_3 \to e_4$. Then $e_4 \leftarrow e_3 \leftarrow h(e_a, e_2)$, so the divergence $e_3 \leftarrow h(e_a, e_2) \to e_4$ is only temporary: the left rewrite is not maximal and can be extended.

Of course, confluence applied in this case is simply an instance of the congruence rule: $h(e_a, e_2)$ is in two E-classes that need to be merged by taking the union of e_3 and e_4, which introduces the rewrite rule $e_3 \to e_4$ or its converse.

3.1.1 The union-find

The above rewrite system represents an E-graph closed under congruence. More generally, besides rules of the form $f(e_1, \ldots) \to e_j$ representing the E-nodes in each E-class, we should consider rules $e_i \to e_k$ representing the equivalence relation defined by the union-find structure. Such a rule expresses that $e_i = e_k$ under this equivalence relation, but also gives us a way to compute a canonical representative of the equivalence class containing e_k by iteratively following the chain $e_i \to e_k \to \ldots$, as long as we maintain the union-find invariant that there are no cycles.

The set of equations derived from the union-find and derived from the E-nodes completely describe the equational theory of the E-graph. Congruence-closure doesn't change the equational theory, but does alter the rewrite system derived from the E-graph to make it more amenable to computing the consequences of the equational theory, in particular by ensuring *confluence*.

3.2 E-graphs simplify confluence

E-graphs in the rewriting perspective consist of two kinds of rules: union-find rules $e_i \to e_j$ and E-node rules $f(\ldots) \to e_k$. To turn an equation $s = t$ into rules, we flatten it completely into E-node rules, and add a union-find rule when the resulting top E-nodes would be in different E-classes.

The advantages of this flattening are twofold. Firstly, termination[1] of the rewriting system is easy to determine: E-node rules make a term smaller, so we just need to ensure there is no infinite cycle among the union-find rules. Secondly, confluence is much easier to check for and restore.

Termination and confluence together give convergence, which means that we can apply the rules in any order to a term and get a unique normal form. If the normal form is e_i, then that term is in the E-graph and equivalent to any other term that also yields e_i. We could also get, say, $f(e_1, e_2)$, which means that the term is not in the E-graph, but its two subterms are.

We will use $\to^*$ for the reflexive transitive closure of the rewriting relation.

Here are the two ways in which an E-graph-like rewriting system could fail to be confluent:

1. $e_1 \to^* e_2$ and $e_1 \to^* e_3$
2. $f(e_1, e_2, \ldots) \to^* e$ and $f(d_1, d_2, \ldots) \to^* d$, where for each i, $e_i \to^* d_i$

The first case is resolved by the union-find. The second case is resolved by congruence-closure.

If, whenever we have the rule $e_i \to e_j$, we replace e_i everywhere in our rewriting system by e_j, then the confluence rule is even simpler, as we only need to check for $f(e_1, \ldots) \to e, f(d_1, \ldots) \to d$, and $e_i = d_i$ or $e_i \to d_i$. Now we don't even have to think about $\to^*$.

So, E-graphs simplify the problem of congruence closure by reducing the problem to checking for certain kinds of *local* non-closure (local failures of confluence), resolving them, and repeating until fixpoint. With this method, there is no added complexity to handling deeply-nested subterms.

Such local failures of confluence are known as critical pairs[2] (see section 3.3.4).

3.3 Formalizing congruence closure

To provide a formal definition of various aspects of E-graphs and EMTs, we will need some basic notions of signatures, terms, and flat terms.

[1] in some sectors of computer science, termination is more commonly known as *strong* normalization

[2] They are also required to be minimal, but we gloss over this here

3.3.1 Terms and signatures

Definition. A **signature** is a set Σ of function symbols with an arity given by a function $\alpha : \Sigma \to \mathbb{N}$. Given a set of constants A, a **flat term** $\Sigma(A)$ is a pair of an element of $f \in \Sigma$ together with a list of elements of A of length $\alpha(f)$. A **term** over A, denoted by $\Sigma^*(A)$, is defined inductively as either a constant $a \in A$ or a flat term over $\Sigma^*(A)$.

This definition builds the notion of term inductively from flat (1-layer) terms, which are useful in their own right.

3.3.2 Relations

A binary relation is some subset of the Cartesian product of two sets, which we will write $\to \subseteq S \times T$. We can write elements of the relation either as $(s,t) \in \to$ or $s \to t$.

If $\to$ is a relation, we will (naturally) write its transpose as $\leftarrow$: $a \to b$ iff $b \leftarrow a$.

There are some natural forms of closure we can apply to relations:

- Given a relation $\to \subseteq S \times S$, its reflexive transitive closure $\to^* \subseteq S \times S$ is the smallest relation containing $\to$ such that $s \to^* s$ for each $s \in S$ and $s \to^* u$ if $s \to^* t$ and $t \to^* u$ for each $s, t, u \in S$.
- Given a relation $\to \subseteq \Sigma(A) \times A$, we can define a generalized notion of reflexive transitive closure as the smallest relation $\to^* \subseteq \Sigma^*(A) \times A$ such that
  1. $a \to^* a$ for each $a \in A$
  2. $f(t_1, \ldots, t_n) \to^* a$ if $t_i \to^* a_i$ and $f(a_1, \ldots, a_n) \to a$ for each f, a, t_i, and a_i

  Informally, $t \to^* a$ if we can reduce the term t to a by reducing from the leaves and repeatedly reducing until it is a single element of A.

  An example of this notion that we have already seen is that if an E-graph with E-node rules $\to_F$ rewrites the term t to e_i, then it assigns the term t to the E-class e_i.
- Given a relation $\to \subseteq \Sigma^*(A) \times \Sigma^*(A)$, its reflexive transitive congruence closure $\to^{*\mathcal{C}} \subseteq \Sigma^*(A) \times \Sigma^*(A)$ is the smallest relation containing $\to$ such that
  1. $a \to^* a$ for each $a \in A$
  2. $f(t_1, \ldots, t_n) \to^{*\mathcal{C}} a$ if $t_i \to^{*\mathcal{C}} a_i$ for each f, a, t_i, and a_i

  This captures the natural notion of applying arbitrarily many rewrites from $\to$ to a term, and allowing the possibility of applying them at any subterm.

  We define some basic properties and terminology for binary relations:

  – A relation $\to \subseteq A \times B$ is *functional* if, for every $a \in A$, there is at most one $b \in B$ such that $a \to b$.

  – If $a \to b$, then we call b a *direct successor* of a.

– If there is no b that is a direct successor of a under $\to$, we say that a is a normal form.

– A relation $\to$ is *terminating* if there is no infinite chain $x \to y \to \ldots$.

3.3.3 E-graph rewrite systems

Definition. An *E-graph rewrite system* over Σ consists of a set C of E-classes, a relation $\to_U \subseteq C \times C$ of union-find rules, and a relation $\to_F \subseteq \Sigma(C) \times C$ of E-node rules.

We can now define closure under the union-find's equivalence relation:

Definition. An E-graph rewrite system is equivalence-closed if the following hold:

- (Functionality) $\to_U$ is functional.
- (Acylicity) There are no cycles $e \to_U e' \to_U \ldots \to_U e$ in $\to_U$
- (E-canonization) If $f(e_1, \ldots, e_n) \to_F e$, then all of $e_1, \ldots, e_n$ and e have no direct successors.

Finally, we can define congruence closure:

Definition. An E-graph rewrite system is congruence-closed if it is equivalence-closed and the relation $\to_F$ is functional. Explicitly: if $f(e_1, \ldots, e_n) \to_F e$ and $f(e_1, \ldots, e_n) \to_F f$, then $e = f$.

3.3.4 Critical pairs

Given two rewrite rules $s_1 \to t_1$ and $s_2 \to t_2$, we can ask if they form a critical pair. Formally, we think of each individual rewrite rule as just an ordered pair, but we keep the $\to$ between the sides as a reminder that each particular pair tends to come from a relation. However, note that it is often reasonable to consider critical pairs between rewrites from two different relations.

Definition. A *peak* between $s_1 \to t_1$ and $s_2 \to t_2$ exists if there is a term s such that both s_1 and s_2 are *overlapping* subterms of s.

The *critical pair* for the peak is the pair of terms obtained by replacing s_1 with t_1 and s_2 with t_2 inside s.

Note that for terms as we have defined them, subterms can only overlap if one is a subterm of the other. When we generalize to AC-critical pairs, we need to work with a more general kind of overlap.

Definition. A critical pair $t_1 \leftarrow_A s \to_B t_2$ is *joinable* if there is a term t such that $t_1 \to_B^* t \leftarrow_A^* t_2$.

3.3.5 Congruence closure is confluence

Lemma 1. *If an E-graph rewrite system is equivalence-closed, then every critical pair between $\to_U$ and $\to_U$ is joinable.*

Proof. Suppose we have $e_1 \leftarrow e \rightarrow e_2$. Then by the functionality requirement for equivalence-closure, $e_1 = e_2$ and so $e_1 \rightarrow_U^* e_1 \leftarrow_U^* e_2$ □

Lemma 2. *If an E-graph rewrite system is equivalence-closed, then every critical pair between $\rightarrow_{UC}^*$ and $\rightarrow_F$ is joinable.*

Proof. Suppose we have $f(d_1, \ldots, d_n) \leftarrow_U^{*\mathcal{C}} f(e_1, \ldots, e_n) \rightarrow e$. We must have, by the definition of $\rightarrow_U^{*\mathcal{C}}$, that $e_i \rightarrow_U^* d_i$. Then by E-canonization we must have that $d_i = e_i$. Therefore, $f(d_1, \ldots, d_n) \rightarrow_F^* e \leftarrow_U^{*\mathcal{C}} e$. □

Lemma 3. *If an E-graph rewrite system is congruence-closed, then every critical pair between $\rightarrow_F$ and $\rightarrow_F$ is joinable. Moreover, every critical pair between $\rightarrow_U \cup \rightarrow_F$ and $\rightarrow_U \cup \rightarrow_F$ is joinable.*

Proof. Every critical pair are already equal by the functionality of $\rightarrow_F$, so they are joinable.

So (F, F) critical pairs are joinable, and (U, F), (F, U), and (U, U) critical pairs are joinable by the previous lemmas for equivalence-closure. □

Theorem 2. *If an E-graph rewrite system is congruence-closed, then the relation $(\rightarrow_U \cup \rightarrow_F)^{*\mathcal{C}}$ is confluent.*

Proof. $\rightarrow_U \cup \rightarrow_F$ is *locally confluent* by the previous lemma. Local confluence is just the property of every critical pair being joinable. Further, $\rightarrow_U \cup \rightarrow_F$ is terminating by the acylicity of $\rightarrow_U$ and the fact that $\rightarrow_F$ maps terms to smaller terms. Therefore, we can apply Newman's lemma [1, Section 2.7]. □

This theorem is fairly intuitive and easy to verify without this level of formality, but these techniques will be helpful for the more involved proofs for EMTs.

3.4 The semantics of equality saturation

Given an identity $\forall x_1 \ldots x_n.\ t_1 = t_2$, we can think of it as two rewrite rules, $t_1 \rightarrow t_2$ and $t_2 \rightarrow t_1$.

We can then extend our notion of critical pair to include substitution for variables. For instance, $x + y \rightarrow y + x$, one half of C, and $e_1 + e_2 \rightarrow e_3$ have the critical pair $e_2 + e_1 \leftarrow e_1 + e_2 \rightarrow e_3$, which uses the substitution instance $x := e_1, y := e_2$.

Definition. A critical pair between the rewrite $s_1 \rightarrow t_1$ and the rewrite $s_2 \rightarrow t_2$ is a substitution σ and a critical pair between $\sigma(s_1) \rightarrow \sigma(t_1)$ and $\sigma(s_2) \rightarrow \sigma(t_2)$.

Using this insight, equality saturation can be also seen to proceed by joining critical pairs, although there is one key restriction: we only look for *grounded* critical pairs: critical pairs where at least one of the rules involved is not an identity. For instance, given the identities $\forall x.\ f(a, x) = b$ and $\forall x.\ f(x, a) = c$, we would derive the two rewrite rules $f(a, x) \rightarrow b$ and $f(x, a) \rightarrow c$, which have a critical pair $b \leftarrow f(a, a) \rightarrow c$.

However, this critical pair, even though it consists entirely of ground terms, is not *grounded*, because it doesn't originate as the critical pair between an identity and a ground non-identity.[3].

After having turned our identities into rewrite rules (containing variables), we can define when an E-graph rewrite system is *saturated* under this set $\to_I$ of rewrite rules:

Definition. An E-graph rewrite system is saturated under R if, for every $t_1 \to_R t_2$, substitution σ, and E-class e_1, then $e_1 \leftarrow_F^* \sigma(t_1)$ implies $\sigma(t_2) \to_F^* e_1$.

This amounts to saying that every critical pair $\sigma(t_2) \leftarrow_R \sigma(t_1) \to^* e_1$ is resolved by $\sigma(t_2) \to^* e_1$.

Formulating equality saturation using critical pairs like this goes beyond the Abstract Congruence Closure framework of [3].

3.5 AC-critical pairs

When we want to reason about AC, we need the notion of AC-critical pair.

Example. If $e_1 + e_2 \to e_3$ and $e_4 + e_2 \to e_5$, then there is a critical term $e_1 + e_2 + e_4$. This is AC-equal to $(e_1 + e_2) + e_4$, which rewrites to $e_3 + e_4$, and also to $e_1 + (e_2 + e_4)$, which rewrites to $e_1 + e_5$. So, our critical pair is $e_3 + e_4$ and $e_1 + e_5$.

In general, we can consider two variadic $+$-terms, $+(e_1, \ldots, e_k)$ and $+(d_1, \ldots, d_i)$. We will think of the arguments as multisets, so that $\{e_1, \ldots, e_k\} = E$ and similarly for D. Let the multiset O contain each element with the minimum multiplicity it occurs in either E or D. For example, if $E = \{e_1, e_1, e_2, e_3\}$ and $D = \{e_1, e_3, e_4\}$, then $O = \{e_1, e_3\}$. Construct the terms $E - O + D$ and $E + D - O$, where $+$ and $-$ are multiset addition and subtraction (that is, they act elementwise on the multiplicities of elements). It is easy to see that $E - O + D = E + D - O$ up to AC, so this will be our peak. O is, by construction, contained in both E and D, so the subtraction is always defined.

Definition. Given two rewrites $+(E) \to t_1$ and $+(D) \to t_2$, their AC-critical pair is $t_1 + D - \min(D, E) \leftarrow E + D - \min(D, E) \to E - \min(D, E) + t_2$, as long as $\min(D, E)$ is not the empty multiset.

The non-emptiness condition on AC-critical pairs corresponds to the condition for ordinary critical pairs that the relevant subterms overlap in the peak term (section 3.3.4).

Note that $E + D - \min(D, E) = \max(D, E)$ by the integer identity $n + m = \max(n, m) + \min(n, m)$, but this form obscures somewhat the structure of the peak term and in particular the overlap term. $E + D - O$ can also be thought of as $(E - O) + (D - O) + O$: we add the non-overlapping parts and then add a single copy of the overlap.

[3]This, besides the use of an extended signature, is the key difference between equality saturation and Knuth-Bendix completion

3.5.1 AC-subterms

An AC-subterm s of a term t is one whose overlap with t is s itself. More concretely, this means that s is a sub-multiset of t. This forms a natural generalization of the ordinary notion of subterm.

3.6 AC-critical pairs and equality saturation

Ordinary (non-EMT) E-graphs handle AC-critical pairs the same way as critical pairs for any other theory, with equality saturation: we *ground* the AC identities by matching either side with the E-graph, and adding the equivalent other side (if it is not present) and merging the equivalence classes corresponding to either side.

Example. Suppose we have the E-node rules $e_1+e_2 \to e_3$ and $e_4+e_2 \to e_5$. Applying the C identity gives $e_2 + e_1 \to e_3$ and $e_2 + e_4 \to e_5$, but the A identity cannot be applied, since that would require either e_3 or e_5, which are on the right of a +-node rule, to also be on the left.

Now suppose we add a +-node rule $e_3 + e_4 \to e_4$. Now the A-rule applies to the term $(e_1 + e_2) + e_4$, which rewrites to e_4, and we add the corresponding right-hand side $e_1 + (e_2 + e_4)$ to the E-graph. Since $e_2 + e_4$ is already present and rewrites to e_5, we add the new +-node rule $e_1 + e_5 \to e_4$.

Given that both $e_3 + e_4$ and $e_1 + e_5$ now rewrite to e_4, the critical pair $e_3 + e_4$ and $e_1 + e_5$ is resolved.

Why do ordinary E-graphs fare poorly with AC? We have seen that equality saturation with AC will typically generate an exponential number of E-nodes. It seems the 'redundant' E-nodes and E-classes play a role in AC-saturation, or at least *could* play a role when viewed through the lens of the general machinery of equality saturation.

Using the terminology of peaks, critical pairs, and overlaps, we can give a succinct description of how AC-saturation behaves and how it differs from AC-closure:

- In ordinary E-graphs with AC-saturation, we eagerly create AC-subterms of every AC-term in the E-graph. Let $\Sigma(M)$ denote any term or E-class representing the sum of the terms in the multiset M. If we represent the terms $\Sigma(E)$ and $\Sigma(D)$ with overlap $\Sigma(O)$, $\Sigma(O)$ will be assigned its own subterm with an E-class $\Sigma(O) \to_F^* o$, and there will be rewrites such that $\Sigma(O) + \Sigma(E - O) \to_F^* e$ and $\Sigma(O) + \Sigma(D - O) \to_F^* d$.

  If there were any E-class representing the peak $\Sigma(E + D - O)$, AC-saturation would eventually add a rule $e + \Sigma(D - O) \to_F p$, and by associativity the rule $d + \Sigma(D - O)$ will then be added. Thus, if this critical pair appears as a (sub)term anywhere, AC-saturation will (eventually) resolve it.

- In EMTs with AC-closure, we do not represent the large number of subterms (potentially infinitely many in the case of cycles, as seen in section 2.5). However, we do need to materialize the actual critical pairs more eagerly: whereas AC-saturation effectively only looks at critical pairs of the form $e_1 \leftarrow t \to e_2 + e_3$,

AC-closure must handle critical pairs of the form $e_1 + \ldots \leftarrow t \rightarrow d_1 + \ldots$ where both terms contain the operator. Handling these critical pairs is essential because one of the critical terms may be a subterm of an existing AC-node, or might *become* a subterm of an AC-node yet to be inserted. Resolving each critical pair allows this subterm relation to be established by later critical pair resolution, since the subterms just form special kinds of critical pairs.

As an example of the last point, consider the rules

$$e_1 + e_2 \rightarrow e_4 \tag{3.1}$$
$$e_2 + e_3 \rightarrow e_5 \tag{3.2}$$
$$e_1 + e_5 + e_6 \rightarrow e_7 \tag{3.3}$$

The critical pair $e_1 + e_5 \leftarrow e_1 + e_2 + e_3 \rightarrow e_4 + e_3$ means that $e_3 + e_4$ and $e_1 + e_5$ are equal, and we potentially need to be able to deduce that $e_7 = e_1 + e_5 + e_6 = e_3 + e_4 + e_6$. The last equation might only appear *later* than the examination of this critical pair, so the critical pair must be resolved into a rule; it is not enough to just scan for current terms with either side as a subterm when resolving a critical pair.

3.7 EMTs: handling AC-critical pairs directly

EMTs for AC add a third class of rewrite rule to E-graphs, the AC-rules. We will focus on the case with one AC-operator $+$, although the generalization to multiple rules is straightforward; the different AC-operators essentially get their own set of AC-rules, and these sets do not interact directly.

Our algorithm for EMT(AC)s is similar to *Buchberger's algorithm* [1, Chapter 8], and is an adaptation of methods described in [3, 19, 9].

3.7.1 A note on our approach

For AC-rules, do we want to allow rules of the form $e_1 + \ldots + e_n \rightarrow d_1 + \ldots + d_k$ or restrict to simply $e_1 + \ldots + e_n \rightarrow e$? The former is closer to usual presentations of Buchberger's algorithm, while the latter is more in the spirit of E-node rules.

We will refer to these as the *side-table* and *integrated* approaches. Unfortunately, there is a difficulty with the integrated approach.

3.7.2 The difficulties of integrated EMT(AC)s

Consider the very simple example

$$a + b \rightarrow_{AC} d$$
$$a + c \rightarrow_{AC} e$$

These rewrite rules overlap on a and thus form a critical pair $c + d \leftarrow_{AC} a + b + c \rightarrow_{AC} b + e$.

Suppose we try to add new rules to resolve the critical pair

$$c + d \to_{AC} f$$
$$b + e \to_{AC} f$$

for a new E-class f. This is in line with E-graphs, where rules always have atomic right-hand-sides.

Unfortunately, the new rules continue to have *unjoinable* critical pairs with the originals:

$$d + e \leftarrow_{AC} a + b + e \to_{AC} a + f \leftarrow_{AC} a + c + d \to_{AC} d + e$$

Notice how both new critical pairs are between $a + f$ and $d + e$. Resolving these in the same way with a new constant g would lead to further unjoinable critical pairs. Both new critical pairs could be joinable if we could turn the $\to_{AC} a + f \leftarrow_{AC}$ in the centre into $\to_{AC} a + f \to_{AC}$.

This problem is solved by the side-table approach which instead inserts the new rule $c + d \to_{AC} b + e$ or its reverse, skipping the unjoinable central term $a + f$.

It is possible to make the integrated approach work with an additional rule $a+f \to_{AC} e + d$. This extra rule has the property that all critical pairs with it where a is the overlap are already joinable, while there are no critical pairs with f as the overlap, since f only appears on the right-hand side of the rules introduced above. Therefore, this rule can function as a 'shadow' rule, available for canonization but that does not need to take part in critical pair calculation.

However, introducing two kinds of rule (normal and 'shadow') in order to make all the normal rules have the standard E-graph form with an atomic right-hand-side seems more complex than its worth.

Prior approaches to congruence closure modulo AC [3, 19, 9] have entirely focused on what we refer to as the side-table approach.

3.7.2.1 Aside: why don't ordinary E-graphs have this problem?

We have seen that the AC-term $a + b$ overlaps with the AC-term $a + c$, and then can overlap with the term $b + e$ in the critical pair. Why can't this happen with ordinary E-graphs?

For ordinary terms, if t is a subterm of $f(s_1, s_2)$, we can conclude that either $t = f(s_1, s_2)$, or that t is a subterm of either s_1 or s_2. For instance, consider the critical pair for the deeply-nested rules $f(g(h(a))) \to s$ and $h(a) \to t$. The critical pair consists of $f(g(t))$ and s. We can reason that if $h(a)$ overlaps $f(g(t))$, it must overlap t already, while if $f(g(h(a)))$ overlaps $f(g(t))$, either it already overlaps t or $t = h(a)$.

By contrast, $a + b$ overlaps $b + e$ without either a being equal to e or $a + b$ being a subterm of one of b and e.

The trivial kinds of overlap for ordinary E-graphs are enough to argue that the 'endless overlap' problem can't happen. On the other hand, for EMTs, non-trivial kinds of overlap require a generalization of the kind of rules allowed.

3.7.3 EMT(AC)s via a side-table, briefly

With the side-table approach, we think of the set of AC rules $e_1+\ldots+e_n \to d_1+\ldots+d_n$ as being a separate thing entirely to the E-graph.

On insertion, we do not add any E-node rules whenever the function symbol is an AC symbol $+$, but generate a new E-class name e and add the AC-rule $e_1 + \ldots + e_i \to e$ when we insert the flattened term $e_1 + \ldots + e_i$.

To compute the AC-closure, we take every pair of two AC-rules $+(R) \to +(S)$ and $+(T) \to +(U)$, where R, S, T, U are multisets of E-classes. Then we can define, as usual,

- The overlap: $\min(R, T)$
- The critical term: $\max(R, T) = R + T - \min(R, T)$
- The critical pair: $c_1 = \max(R, T) - R + S$ and $c_2 = \max(R, T) - T + U$

When the overlap is not the empty multiset, then we take the critical pair and try to rewrite both sides by existing rules until it is minimal. If it is still non-trivial, we *orient* it, according to some ordering on flat AC-terms. A common choice for Buchberger's algorithm is *grevlex* order, graded reverse-lexicographic, which is used in our implementation. It has been observed that the performance of Buchberger's algorithm depends quite critically on the ordering used [5, Chapter 2, section 10, subsection *Complexity issues*]. Unfortunately, there is no (mathematically) canonical ordering to choose.

After orienting the reduced rule, we reduce every other rule with respect to it, and add it to the set of AC rules. This interreduction step is a standard step in Buchberger's algorithm, but may not be natural from the E-graph viewpoint, as it is a form of destructive rewriting on the E-graph.

The final ingredient is the 'combination rule'. If we were ever to have a reduced rule $e_1 \to e_2$, with 1-term sums on either side, we should instead treat this as a union-find rule, and correspondingly propagate its consequences into the union-find and E-node rules. And vice versa, whenever the union-find rules are updated, we need to update any AC-rules according to the new ordering on atoms induced by the union-find rule.

3.8 EMT(AC)s, formally

Definition. An EMT(AC) consists of the set C and the relation $\to_U$ and $\to_F$ of an E-graph together with a relation $\to_{AC} \subseteq \mathcal{M}_{\text{fin}}(C) \times \mathcal{M}_{\text{fin}}(C)$

3.8.1 Canonization

First, we describe AC-canonization:

Algorithm 1. Input: an EMT $(C, \to_U, \to_F, \to_{AC})$ and an AC-term t.

Procedure: repeat either of the following steps until neither is applicable:

1. If there is a rule $s \to_{AC} e_i$ such that s is an AC-subterm of t, replace t with $t - s + e_i$ ($t - s$ is defined as s is an AC-subterm of t).
2. If there is a rule $e_j \to_U e_i$ such that e_j appears in t, replace all copies of s_j in t with e_i.

3.8.2 The algorithm for EMT(AC) closure

Algorithm 2. Input: an EMT $(C, \to_U, \to_F, \to_{AC})$ and a set R of AC-equations.

Procedure: while R is not empty:

1. Remove $t_1 = t_2$ from R.
2. AC-canonize both t_1 and t_2. Call the canonized forms t_1' and t_2'.
3. If $t_1' = t_2'$, then go to the next iteration.
4. If t_1' and t_2' are single E-classes e_1 and e_2, then:
   (a) Add $e_1 \to e_2$ to $\to_U$ and recompute congruence closure on $\to_U$ and $\to_F$.
   (b) For any AC-rule $s \to_{AC} e_i$ where one of the E-classes in s or e_i has a successor in the union find, remove that rule from $\to_{AC}$ and add $s = e_i$ to R.
5. Otherwise:
   - Compare the terms t_1' and t_2' according to some ordering, and call the smaller t_a and the larger t_b.
   - Add the rule $t_b \to_{AC} t_a$ to $\to_{AC}$.
   - For every rule $t_l \to_{AC} t_r$ except the rule just added, if t_l has t_b as an AC-subterm, remove it from $\to_{AC}$ and add $t_l = t_r$ to R.
   - For every rule except the one just added, compute the critical pair between that rule and the new rule, and add the equation $c_a = c_b$ to R, where c_a and c_b are the terms of the critical pair.

3.8.3 Correctness of the algorithm

Define the *Dickson ordering*[4] on multisets as the partial ordering $s \geq t$ if the multiplicity of every element in s is greater than or equal to the multiplicity of the

[4] the name 'Dickson ordering' for this common ordering is taken from Kapur [9, 10]

same element in t. This is the same as saying that s has t as an AC-subterm, or that s is a super(multi)set of t.

Lemma 4. *At each iteration of algorithm 2, there are no rules in $\rightarrow_{AC}$ with left-hand-sides that are related by $\geq$ in the Dickson ordering.*

Proof. If $s \geq t$, then s has t as an AC-subterm. Before each iteration, this invariant holds for every pair of existing rules. When a new rule is added in step 5, any rule which would be greater than or equal to the new left-hand side is removed and added to R, so this invariant is restored. Conversely, both sides of the new rule are AC-canonized with respect to existing rules, so its left-hand side cannot be greater than or equal to the left hand side of any of them. □

Lemma 5. *(Stability) When a rule $t_l \rightarrow_{AC} t_r$ is removed, any further rule with a left-hand-side s which would satisfy $s \geq t_l$ will satisfy this with one of the remaining rules.*

Proof. Whenever we remove a rule with left-hand-side t_l, this is because we determined that $t_l \geq t$ for some new left-hand-side t. So any s for which $s \geq t_l$ would then satisfy $s \geq t_l \geq t$ with the remaining set of rules that include t on the left of the new rule. □

Therefore, the set of left-hand-sides in $\rightarrow_{AC}$ is a set of elements in $\mathcal{M}_{\text{fin}}(C)$, none greater than any other in the Dickson ordering. Each iteration, we either

- Add a new rewrite to $\rightarrow_U$, and do some other changes
- Add a new element to this set of left-hand-sides, possibly removing some existing elements, and do some other changes
- Remove one equation from R.

Since C is fixed, only finitely many changes to $\rightarrow_U$ are possible, since at most we can equate every element in C in the union-find.

Once the algorithm reaches the point it doesn't alter $\rightarrow_U$ any further, then we look at how it changes the set of left-hand sides of rules in $\rightarrow_A C$. Any new element must not just be not greater than any existing element, but also any past ones, by the stability lemma. Therefore, we can consider the entire set of previous and current elements, and this set only grows.

By Dickson's lemma [2], there can't be an infinite set of elements in $\mathcal{M}_{\text{fin}}(C)$ over a finite set C that are pairwise incomparable. Therefore, the process of adding rules must terminate at some point.

Once adding rules stops, the remaining equations in R will be removed and the algorithm terminates.

Correctness is fairly straightforward: critical pairs are added to R and eventually resolved. Since we add all critical pairs between new rules and existing ones, the final system of AC-rewrites cannot have critical pairs that are unjoinable.

The properties of the AC-term ordering come into play in proving that *canonization* terminates. In particular, we can reuse the Dickson-order argument, as long as the term order is reflexive, antisymmetric, and transitive (and thus actually an order!), and is an extension of the Dickson ordering, so that if $t \leq s$ in the Dickson ordering, then $t \leq s$ in the term ordering. With these properties, the iterations of the canonization cannot increase the term in the Dickson ordering, and so can only rewrite the term within the finite set of of terms that are not Dickson-greater than the original. Then, we rule out cycles by the ordering properties: if $t_1 \geq t_2 \geq \ldots \geq t_1$, then $t_1 \geq t_2 \geq t_1$ and so $t_1 = t_2$, which is only possible if the rewrite rule involved is of the form $s \rightarrow_{AC} s$. However, the algorithm never adds rules with equal left-hand and right-hand sides to the rewrite system.

4

Beyond equations: E-matching, E-analysis, and E-xtraction

Often, applications of E-graphs don't just use them as an equational theory solver, but involve running certain analyses on the E-graph (perhaps to allow conditional rewrites) or want to extract the smallest term representing an equivalence class, according to some cost metric.

Besides that, E-matching is a core part of the equality saturation loop.

The notion of E-class analysis was first introduced in [23, Section 4.1]. E-class analyses, a robust implementation of E-matching, and a framework for extraction are some of the key pieces making the `egg` implementation of E-graphs so useful [23].

We need to generalize all of these to EMTs. Thankfully, the 'three Es' are quite similar, and can be generalized together. We begin by describing the three Es in the setting of ordinary E-graphs.

4.1 E-analysis

Suppose we have, for each function symbol f of arity $\alpha(f)$, an *interpretation* of that function symbol over a domain D, so that if $d_1, \ldots, d_{\alpha(f)} \in D$, then the interpretation $[\![f]\!]$ maps $(d_1, \ldots, d_{\alpha(f)}) \in D^{\alpha(f)}$ to $[\![f]\!](d_1, \ldots, d_{\alpha(f)}) \in D$.

This interpretation gives us a basis for an E-analysis. Keep a map a from E-class names to values in D, and when we insert $f(e_1, \ldots)$ into a new E-class e, set $a(e) = [\![f]\!](a(e_1), \ldots)$.

We need a binary operation $\sqcap : D \times D \to D$ on the domain, which merges annotations when their E-classes are merged. To make this process well-defined regardless of the details of the congruence-closure/rebuilding process, we want $\sqcap$ to be associative, commutative, and idempotent.

4.2 Extraction

Suppose we have some cost function $c : T \to K$ mapping terms $t \in T$ to a totally-ordered set C. Then we can define extraction as an analysis by taking our domain to be $T \times C$, and defining

$$(t_1, c_1) \sqcap (t_2, c_2) = \textbf{if}\, c_1 > c_2\, \textbf{then}\, (t_1, c_1)\, \textbf{else}\, (t_2, c_2)$$
$$[\![f]\!]((t_1, c_1), \ldots) = (f(t_1, \ldots), c(f(t_1, \ldots)))$$

For this to be well-defined, we require that cost is monotonic: if $c(t_i) < c(s_i)$, then $c(f(t_1, \ldots)) < c(f(s_1, \ldots))$. Otherwise, lower-cost child terms could make higher-cost parent terms, so we cannot rely on a simple propagation strategy to find the overall lowest-cost term in each E-class.

4.3 E-matching

4.3.1 Compiling E-matching to an E-analysis

Suppose we have some pattern, like $f(x, g(a, x, y), h(y))$, where x and y are variables. The first step is to compile the patterns into a *match table*, which for this example would look like

- Leaf variables: $L = \{x, y\}$
- States:

$$\begin{aligned} a &\to s_1 \\ g(s_1, \bot(x), \bot(y)) &\to s_2(x, y) \\ h(\bot(y)) &\to s_3(y) \\ f(\bot(x), s_2(x, y), s_3(y)) &\to s_4(x, y) \end{aligned}$$

- Final state: s_4

The idea is that, thinking of the pattern term as a tree, we can assign a state to each node in the tree. Then, our E-analysis for doing E-matching will map E-classes to the domain of sets of pairs of the form (state, set of substitutions). A state s with substitution $\{x \mapsto e_i\}$ is represented as $s(e_i)$, since each state has a fixed list of previously-matched symbols that appear in its associated substitutions.

For example, if our E-analysis for E-matching on the above pattern assigns to some E-class e the set $\{(s_1, \{\}), (s_2, \{x \mapsto e_1, y \mapsto e_2\}), (s_2, \{x \mapsto e_3, y \mapsto e_1\})\}$, then the E-class e has matched the sub-pattern a (as this set contains s_1) and the subpattern $g(a, x, y)$ in two ways: once via $g(a, e_1, e_2)$ and once via $g(a, e_3, e_1)$. Because a and g are different symbols, this set of matches must be the result of e containing the E-node a and at least one E-node of the form $g(-, -)$, and the matches for each E-node in e have been unioned together.

As an aside, E-graphs are closely related to tree automata [18], and our match-table-based pattern matching presents a tree automata defining the terms to be matched. E-matching essentially amounts to computing the intersection of the match table automata with the E-graph automata.

Our interpretation for each function symbol is

$$[\![f]\!](d_1, \ldots) = \{(\bot, \{l \mapsto e\}) \mid l \in L\} \cup \text{match}_f(d_1, \ldots)$$

where

- e is the E-class assigned to the E-class we are analyzing. The interpretation depending on the E-class means that it doesn't quite fit in the framework for E-analysis from above, but this is not a significant change.[1]
- match_f is defined in a way dependent on the match table, taking *joins* across substitutions in the appropriate way when the states match with some entry in the match table. For instance, for the above match table,

$$\begin{aligned}\text{match}_f((s_a, \sigma_a), (s_b, \sigma_b), (s_c, \sigma_c)) = \\ \{(s_4, \{x \mapsto \sigma_1(x), y \mapsto \sigma_2(y)\}) \mid \\ s_a = s_1, \\ s_b = s_2, \\ s_c = s_3, \\ \sigma_1(x) = \sigma_2(x), \\ \sigma_2(y) = \sigma_3(y)\}\end{aligned}$$

For this E-analysis, $\sqcap$ takes the union of the sets of states, and when both sides of the union contain the same state, the sets of substitutions are merged.

4.3.2 Joins and join plans

The above scheme for E-matching is not the only approach. E-matching is a case of evaluating certain kinds of *conjunctive queries* over the E-graph thought of as a database [25], and there are many ways to evaluate such queries [21]. We leave describing other evaluation strategies for E-matching in the context of EMTs to future work (section 8.1.2).

4.4 The 'Three Es' for EMTs

Generalizing to EMTs has some subtleties. Part of our motivation for EMTs is to avoid explicitly representing nodes that would be present in the E-graph if running purely with equality saturation. Since an E-analysis is defined per-E-class, our EMT E-analysis will not *directly* be defined for certain E-classes that simply do not exist in our EMT.

[1] It is possible to have an 'Id-counter' as part of the domain and entirely reconstruct a set of E-class ids as a pure E-analysis independently of the ones actually used in the E-graph, but this feels more like a theoretical trick than something useful. $\sqcap$ for this domain requires a lot of remapping of E-ids, for instance

4.4.1 Representation status of terms

Standard E-graphs using AC-saturation will represent all AC-subterms of every term in the E-graph explicitly. For EMTs, one of the key goals is to avoid representing all of these subterms, at the cost of handling AC-critical pairs. For a term t, we will define its *representation status* in an EMT as follows:

- Rewrite t according to all appropriate AC-rules until reaching a normal form.
- If t is now a single E-class id e, t is *directly represented*.
- If t is now a term $e_1 + e_2 + \ldots$, then see if t is an AC-subterm of a term appearing on either side of an AC-rule. In this case, it is *indirectly represented*.
- Otherwise, it is not represented.

4.4.2 AC-analyses

Regardless, we can now spell out what is required for a well-behaved AC-analysis. We want to treat directly and indirectly represented terms on somewhat equal footing, as indirectly represented terms would be (directly) represented in the corresponding E-graph with AC-saturation.

For a directly represented term, we want a variadic interpretation function for any AC-symbol $[\![+]\!](d_1, \ldots)$ that is associative and commutative, naturally. When constructing an AC E-node, we can use this interpretation to assign the analysis.

For indirectly represented terms, we can apply the interpretation function lazily: for instance, if $e_1 + e_2$ is only indirectly represented, we do not store $a(e_1 + e_2)$ in any sense, but compute $a(e_1 + e_2)$ by canonizing $e_1 + e_2$ to $t = e_i + \ldots$ and then calculate $[\![+]\!](a(e_i), \ldots)$ when needed. Since this term would be directly represented in an E-graph with AC-saturation, we ought to consider its annotation to nonetheless not be simply undefined.

A natural worry is that we might have, say, $e_1 + e_2$ and $e_3 + e_4$, that are both indirectly represented, but that ought to be equal: that is, if they were to be directly represented, they would be assigned the same E-class. In that case, both $a(e_1 + e_2)$ and $a(e_3 + e_4)$ ought to be equal to $a(e_1 + e_2) \sqcap a(e_3 + e_4)$. But this is exactly the problem that AC-completion solves! In particular, if $e_1 + e_2 = e_3 + e_4$ according to the equivalence induced by the EMT, then $e_1 + e_2 \to^* t \leftarrow^* e_3 + e_4$. Thus, $a(e_1 + e_2) = a(t) = a(e_3 + e_4) = a(e_1 + e_2) \sqcap a(e_3 + e_4)$.

4.4.3 AC E-matching: EMTs are no panacaea

Porting our 'matching engine' from section 4.3.1 to handle AC-symbols is possible, but it does force us to confront the unavoidable difficulties of AC-matching.

Consider a naïve matching table for a small example pattern, $f(x + x + g(y + z))$.

$$\begin{aligned} \bot + \bot &\to s_1 \\ g(s_1) &\to s_2 \\ \bot + \bot + s_2 &\to s_3 \\ f(s_3) &\to s_4 \end{aligned}$$

Now consider matching this against $f(a + b + g(a + b) + b + a)$.

$g(a + b)$ matches up to state s_2, but we have missed the commutativity of the arguments: not only is $\{y \mapsto a, z \mapsto b\}$ a valid substitution, but also $\{y \mapsto b, z \mapsto a\}$.

Going further, there is another issue: We have a 5-argument $+$ in our input term, but only a 3-argument rule to reach state s_3.

The key to solving these issues is to consider *flat AC-matching*, which is somewhat simpler than the general notion of AC-matching [15, Chapter 8]. In particular, when matching $x + y + \ldots$ against $e_1 + e_2 + e_3 \ldots$, we have to consider every multiset partition of $\{e_1, e_2, e_3, \ldots\}$ into non-empty multisets: so, $\{x \mapsto e_1, y \mapsto e_2\}$ and $\{x \mapsto e_3, y \mapsto e_1 + e_2\}$ are both valid substitutions from the single AC-match.

In our example, $a + b$ can be rewritten via the first match rule to s_1. Since it AC-matches the rule in two ways, we generate both $\{y \mapsto a, z \mapsto b\}$ and $\{y \mapsto b, z \mapsto a\}$ as substitutions for state s_1.

Then $a+b+s_2+b+a$ matches against our third rule. Here, we generate a substitution for x that is $\{x \mapsto a + b\}$: an example of a substitution that mentions a whole *term*, since the term $a + b$ is not necessarily directly representable in the E-graph. We have to filter out all AC-matches to the third rule to the ones that map the two occurences of x to AC-equal terms, as usual.

The upshot is that AC-matching on EMTs is similar to ordinary E-matching on E-graphs, in the sense that it can be broken down to atomic, E-class/E-node-at-a-time steps. However, we have to use single-level AC-matching and AC-equality instead of single-level matching and trivial equality, and our substitutions are not pure E-class ids, but 'indirectly-represented' terms.

In particular, single-level AC-matching is quite expensive: matching $x + y + z + w$ against $a + b + c + d + e$ generates $4 \cdot 5!/2 = 240$ possible substitutions.

One thing to consider in future work is that a matching engine might allow patterns with certain constraints, such as $x + y + z + w, x \leq y \leq z \leq w$, to cut down the search space. In particular, a pattern like $x + y + z + w$ is likely to come up in a context where the goal is to find instances of four things added together, without caring about full space of all substitutions.

4.4.3.1 AC E-matching on cyclic terms

Consider matching the pattern $a + x$ against the EMT representing the equation $a+b = a$. Clearly, $\{x \mapsto b\}$ is one valid substitution, but $a+b = (a+b)+b = a+(b+b)$,

so $\{x \mapsto b + b\}$ ought to be another substitution. In general, we would expect, for every n, a term of n bs added together to be a valid substitution for the match.

The algorithm in the previous section will only find the single match $\{x \mapsto b\}$.

We will sketch out an algorithm that not only handles these cycles, but also can represent the infinite set of matches compactly.

Writing our pattern $a + x$ as a match table gives

$$a + \bot(x) \to_M s_1(x)$$

with $L = \{x\}$ and final state s_1. We use $\to_M$ for the match table relation.

Our AC-rewrite system is just

$$a + b \to_{AC} a$$

Putting these together, we have an AC-critical pair between $\to_M$ and $\to_{AC}$:

$$a + x \leftarrow_{AC} a + b + x \to_M b + s_1(x)$$

Now, $a + x \to_M s_1(x)$, but then we cannot make further progress. In order to make this critical pair joinable, we can introduce $b + s_1(x) \to_{AC} s_1(x)$. This final rewrite witnesses that any match in state s_1, when part of a term containing a b, can form another match in state s_1, since the b can be expanded to $a + b$.

This rewrite-based perspective on matching (and analysis in general) is promising, but we leave the full development to future work (section 8.1.3).

5

Implementation and evaluation

We have built a proof-of-concept implementation of EMT(AC) in Haskell. We discuss some design choices and tradeoffs we made, which could be improved upon in future work, and compare equality saturation against AC-completion running on our implementation.

The implementation can be found at `https://github.com/sofia-m-a/egraphs_mod_theories`.

5.1 Signatures

We want to be somewhat generic in the signature used to build terms. Our EMT is generic over a type constructor `enode` belonging to the following typeclass

```
class (Ord (ACSymbol enode),
       Ord (Symbol enode),
       Ord (enode EId),
       Traversable enode) => Signature enode where

  type Symbol enode

  symbolOf :: enode a -> Symbol enode
  default symbolOf :: (Symbol enode ~ enode ()) =>
                      enode a -> Symbol enode
  symbolOf = void

  type ACSymbol enode

  acSymbolOf :: enode a -> Maybe (ACSymbol enode)
  default acSymbolOf :: (ACSymbol enode ~ Void) =>
                        enode a -> Maybe (ACSymbol enode)
  acSymbolOf _ = Nothing

  reconstruct :: Symbol enode -> [a] -> enode a
  default reconstruct :: (Symbol enode ~ enode ()) =>
                         Symbol enode -> [a] -> enode a
  reconstruct s as = s & unsafePartsOf traverse .~ as
```

```
reconstructAC :: ACSymbol enode -> [a] -> enode a
default reconstructAC :: (ACSymbol enode ~ Void) =>
                ACSymbol enode -> [a] -> enode a
reconstructAC = absurd
```

The idea is that `enode a` is akin to the definition of $\Sigma(A)$ given in section 3.3.1. In particular, `enode` is expected to be all of a `Functor`, `Foldable`, and `Traversable`. All these classes can be derived automatically with relevant language extensions, and they provide useful functionality to work with the `EId`s within an `enode EId`. Moreover, we can define the type of (non-single-layer) terms simply as `Fix enode`, or terms potentially with `EId` leaves as `Free enode EId`, using the standard `data-fix` and `free` packages.

The type family `Symbol` is defined with the intent that `Symbol enode` is isomorphic to `enode ()`. However, the default definition can be overriden to allow more compact definitions than the generic `enode ()`. Note that an enode with `()` in every argument slot is only distinguished by the function symbol, which motivates this definition.

The type family `ACSymbol` is intended to capture the subset of AC function symbols, and comes with partial function `acSymbolOf` to determine if an E-node is headed by such a symbol. Unlike `Symbol`, AC-symbols are treated as inherently variadic. The default is that no symbols are treated as AC symbols.

`reconstruct` and `reconstructAC` build a term from a function symbol and list of arguments. The default implementation of `reconstruct` is unsafe, and relies on the number of arguments to match the arity of the symbol, or it will panic. `reconstructAC` is intended to be safe as AC-symbols are meant to be variadic. `reconstruct` is not actually used anywhere currently, and is just provided for symmetry with `reconstructAC`, which is used to build new AC-terms when finding critical pairs.

Here is an example E-node type constructor implementing `Signature`:

```
data MyENode a
        = F a
        | G a a
        | Constant Int
        | Plus [a]
        | Times [a]
        deriving (Eq, Ord, Show,
                Functor, Foldable, Traversable)

data ACSym = PlusSym | TimesSym

instance Signature MyENode where
  type Symbol MyENode = MyENode ()
  type ACSymbol MyENode = ACSym

  -- symbolOf: default definition is fine
```

```
acSymbolOf (Plus _) = Just PlusSym
acSymbolOf (Times _) = Just TimesSym
acSymbolOf _ = Nothing

-- reconstruct: default definition is fine

reconstructAC PlusSym as = Plus as
reconstructAC TimesSym as = Times as
```

5.2 Implementation tradeoffs

5.2.1 Persistent data structures

We use standard persistent ordering-based data structures from the `containers` library: `IntMap, IntSet, Map`, and `Set`. Using persistent hash-based data structures from the `unordered-containers` library might be faster, but we decided not to worry about the potential constant-factor improvements. It is possible to get *asymptotic* improvements for some parts of the algorithm by switching to non-persistent data structures, such as arrays and mutable hash-tables: this removes the $\log(n)$ overhead of persistence. However, as a proof-of-concept, we decided persistent structures are somewhat easier to work with as they do not require one to thread unique mutable instances everywhere.

5.2.2 Binarization and incremental congruence closure

Downey-Sethi-Tarjan's algorithm for congruence closure [6] improves upon other algorithms such as Nelson-Oppen's algorithm [12] by *binarizing* the function symbols. For instance, given a ternary symbol f where we might insert E-nodes of the form $f(e_1, e_2, e_3)$, we can instead replace it with a two symbols f_2 and π so that we insert the *term* $f_2(e_1, \pi(e_2, e_3))$ instead. This generates multiple E-nodes per original E-node, but it restricts the arity to at most 2, which makes certain kinds of reverse lookup fast: for each E-class id e, it can either occur in the left part of an E-node $h(e, -)$ or in the right part $h(-, e)$, for which we only need to store one other E-class id and symbol.

Such reverse lookups are important for making congruence closure incremental: when we union e with e', we want to lookup the *uses* of e and replace them with e', possibly discovering more unions to carry out in the process.

The egg library [23] does not use either binarization or indeed any sort of incrementality in propagating equalities, instead preferring to batch the congruence closure computation into big 'rebuilding' steps between multiple steps of equality saturation.

Our library does calculate congruence closure incrementally, but we decided for simplicity to omit the binarization trick for congruence closure. Implementing it adds some complexity, and it seems hard to make a form of binarization that works well for the AC part of the system.

5.2.3 Union-find laziness

The union-find structure can achieve an amortized time complexity of $O(n\alpha(n))$ for n union and find steps, where α is the inverse Ackermann function. This is typically achieved by some degree of laziness on union: union does not entirely canonicalize the structure, and instead find operations mutate the structure to compress indirect paths $e_1 \to e_2 \to \ldots$ as they are discovered.

For congruence closure, and thus E-graphs, the laziness possible in the union-find seems difficult to make use of, as we need to detect an instance of congruence of the form

$$f(a_1, \ldots, b, \ldots, a_n) = f(a_1, \ldots, b, \ldots, a_n)$$

once b and c have been equated; it is not enough to just 'discover' that $b = c$ when calling find on b or c or one of their equivalents, as such a find call is not guaranteed to happen at any particular point before we need to discover the congruence.

5.2.4 Non-incrementality of AC

Unlike congruence-closure, for which it is possible to incrementally restore the congruence-closure invariant with a minimal amount of work when changes are made to the E-graph, AC-closure seems hard to make incremental. For instance, suppose we have some ordering on AC-terms, and we union e_i and e_j so that $e_i \to e_j$ becomes a union-find rule. Consider an AC-rule $e_1 + e_2 + e_i + e_3 \to e_4$. We can keep track of the 'uses' of e_i, and thus we know when merging e_i we need to update this rule, but depending on the relative ordering of e_i and e_j, the ordering of the left-hand-side of this AC-rule could change dramatically.

This is one of the difficulties of applying some binarization to AC-symbols: we could store the AC-rules in some kind of trie, but the structure is upended fairly arbitrarily on unions, and then we potentially have to interreduce other rules with the new one, and so forth. Thus any sort of trie-like binarized structure does not seem to help much when computing AC-closure.

In section 6.3.2 we note that AC-closure is ESpace-complete. However, this leaves room for constant-factor or even polynomial-factor improvements in the algorithmic steps of lookup, finding critical pairs, resolving critical pairs, and so forth. If certain parts of these steps can be made incremental, that is a promising path to achieving such improvements.

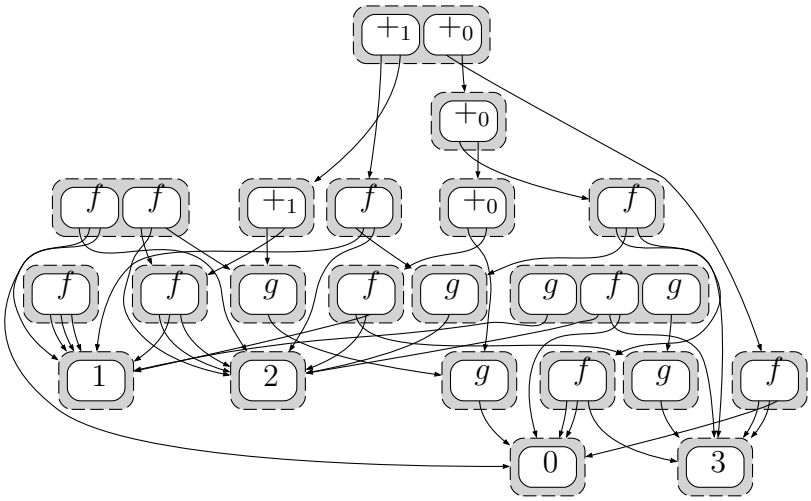
5.3 Evaluation

To evaluate our proof-of-concept implementation, we generate a variety of 'random theories'

- One or two AC-operators ($+_0$, $+_1$)
- A handful of function symbols of fixed random arity

- Some random identities between non-AC terms (as we have yet to fully implement AC-matching)
- If we are testing E-graphs with equality saturation for AC, we generate the A and C identities for each AC-operator

We then generate a number of random starting equivalence classes, each containing some random terms. A typical 'random E-graph' from our distribution looks like this:



Then, we run to saturation under the identities, maintainin AC-completion for our EMTs.

We use a fairly simple size metric: the sum of the number of E-classes, the number of E-nodes, the number of AC-rules, and the sizes of all AC-rules (where the size is the sum of the arities of both sides; so $e_1 + e_2 \to e_3 + e_4 + e_5$ has size 5).

This roughly corresponds to the actual memory size of the E-graph. We choose it over alternatives such as simply counting E-classes which could potentially hide the full size of the AC-rules.

Running a number of random theories and random graphs, we obtain the following results for E-graphs vs EMTs:

This represents 4228 data points. We had a timeout to deal with the possibility of saturation not being reachable for identities which are randomly generated, which is the source of the non-round number.

We can see that for small input data, the AC-completion can sometimes introduce overhead beyond what an E-graph with equality saturation would have, but this quickly disappears as the size of the saturated E-graph increases, and we get significant space savings.

Determining time savings is somewhat difficult for our implementation, as we would need to more carefully instrument the different parts of the propagation and saturation to, for instance, separate out the time used for E-matching. E-matching on the non-AC identities is a significant part of the time for both EMTs and E-graphs in

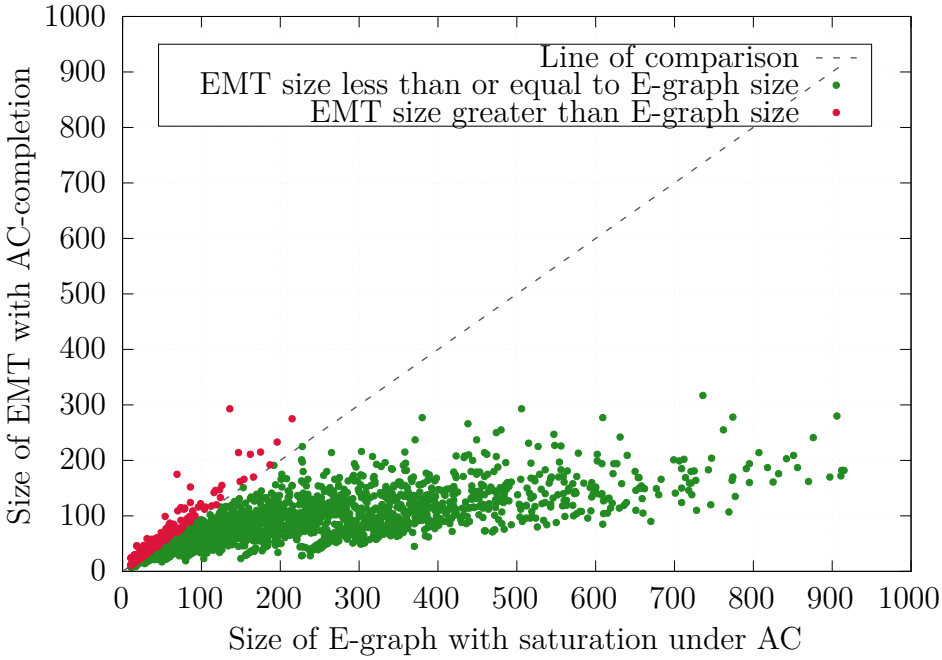


Figure 5.1: Size comparison between EMTs and E-graphs for random E-graphs using AC-operators

our implementation, and thus our fairly slow implementation of matching renders wall-clock times less relevant as a point of comparison.

6

EMTs for other theories

6.1 Common identities

AC is a good example of a theory which E-graphs handle poorly. However, this is not necessarily the case for every common theory, so it may not always be worth the extra complexity of using EMTs. We list some common identities which are worth analyzing:

Theory name	Abbreviation	(Function symbol, arity)	Theory
Associativity	A	$(f, 2)$	$\forall xyz.f(x, f(y, z)) = f(f(x, y), z)$
Commutativity	C	$(f, 2)$	$\forall xy.f(x, y) = f(y, z)$
Unitality	U	$(f, 2), (e, 0)$	$\forall x.f(e, x) = x$ $\forall x.f(x, e) = x$
Distributivity	D	$(f, 2), (g, 2)$	$\forall xyz.g(x, f(y, z)) = f(g(x, y), g(x, z))$ $\forall xyz.g(f(y, z), x) = f(g(y, x), g(z, x))$
Idempotence	I	$(f, 2)$	$\forall x.f(x, x) = x$
Inverse	N	$(f, 2), (i, 1), (e, 0)$	Unitality and $\forall x.f(i(x), x) = e$ $\forall x.f(x, i(x)) = e$
Involution	V	$(i, 1)$	$\forall x.i(i(x)) = x$

Table 6.1: Definitions of some common identities

We use the convention of naming combined theories by taking the juxtaposition of the identity names: as we have seen, AC is the theory with both A and C identities (for some function symbol), and so forth.

This list of identities covers a wide range of common mathematical structures, such as semigroups, monoids, groups, semirings, rings, modules, semilattices, lattices, Boolean algebras, as well as some less common ones such as bands, quasigroups, loops, and near-semirings.

Note that the condition $x \neq 0$ for x to have an inverse in a field is not formalizable in purely algebraic terms, but this list covers every other field identity. A similar observation applies to vector spaces.

6.2 How common theories fare with ordinary E-graphs

In general, using theories with equality saturation will lead to a growth in the size of the graph, as more nodes will need to be seeded to ensure that all consequences of the identities are discovered. We will analyze typical theories and how they fare.

The possibility of cyclic graphs is a core sticking point for how certain theories fare when used in the equality saturation process.

6.2.1 A quick analysis

For C, if we have an E-node $f(e_1, e_2)$, we might need to add the E-node $f(e_2, e_1)$. However, we never need to add a new E-class. Indeed, we never need to add a new E-class for any identity unless one side of the identity contains a *proper* subterm not contained on the other. This property is also satisfied by I. U requires us to possibly add an E-class to represent e, but once this has been added, no new E-classes are generated by the identities.

V generates a new E-class $i(e)$ for each existing E-class e, but then continuing this process for $i(e)$ generates $i(i(e))$ which is already equal to e. In other words, saturation doubles the number of E-classes and goes no further.

The prime examples of 'difficult' identities that we will cover here are, then, A, D, and N.

6.2.2 Weak term acyclicity

[18] gives a definition of when a theory[1] is 'weakly term acyclic', and proves that such theories can be solved with equality saturation in polynomial time. This is a generalization of our basic syntactic analysis of the identities to identify which E-nodes can potentially be created by saturating under an identity.

The intuition is that one can track the 'source' of new E-classes that can be generated during saturation. If, for instance, every three existing E-classes with some particular relation between them generate a fourth potentially new E-class that *can not* be in this same relation with any other classes, we can bound the number of E-classes created as $O(n^3)$ where n is the number of initial E-classes.

6.2.3 The case of N

N is an interesting case because the variable x is only bound on one side of the identity. In other words, if we have an E-class e' representing the constant e, for what values of x should we instantiate $f(i(x), x)$ with to add to the class e'?

[1] actually, they define saturation under a set of rewrite rules, rather than equations, but we can treat equations as bidirectional rewrite rules

Our notion of critical pairs (section 3.3.4) gives a straightforward answer. The pseudo-rule $f(i(x), x) \to e'$ is only relevant insofar as it creates critical pairs with some other existing rule in an E-graph rewrite system. An exact analysis of when this occurs is beyond the scope of this section, given that it depends on what other identities we have in the theory, but the general point is that x will only need to be instantiated with some 'known' E-class.

6.3 The word problem

Doing equality saturation on a theory T, or coming up with a notion of EM(T), connects to the *word problem*.

The word problem is as follows: given some theory T and some equations, does a particular (ground) equation $t_1 = t_2$ hold?

Restricting the set of identities, we get the word problem for commutative semigroups (when T = AC), the word problem for monoids (when T = AU), etc.

6.3.1 E-graphs, EMTs, and the word problem

We can reduce the word problem to equality saturation. If we have some input equations, we can add them to an E-graph, saturate under the identities in T, and then check if t_1 and t_2 are represented in the E-graph by the same E-class. Therefore, equality saturation is at least as hard as the word problem.

Since EMTs are trying to solve the same problem as E-graphs with equality saturation, just in a more efficient way, any good notion of EM(T) for a theory T essentially includes a solver for the word problem over T.

Going the other way, we can partially reduce equality saturation to the word problem. Suppose we have some input terms and equations. We can use a word problem solver to determine if any pair of terms is equal given the equations and the identities in T, and construct an E-graph assigning all equal terms to the same class. This is not necessarily the same E-graph that would result from equality saturation; equality saturation might create all sorts of intermediate E-classes and E-nodes that are not necessarily present in our E-graph derived from the word problem. Indeed, the observation that a word-problem solver can have some internal method for solving the word problem that doesn't do all the intermediate steps of equality saturation is the key observation that makes EMTs worth pursuing.

6.3.2 Results on various theories

For A, the theory of semigroups, the word problem is undecidable [13], and this extends to AU, the theory of monoids.

Therefore, while EMT(A) and EMT(AU) might be worthwhile notions to explore, there is no terminating "A-completion" that can be applied: we can provide infras-

tructure to find and resolve critical pairs incrementally but never have a guarantee that process will terminate.

Adding commutativity is the key ingredient that makes the word problem decidable. The word problem for AC and ACU (commutative semigroups and commutative monoids) is ESPACE-complete [11], which provides a fairly tight bound on the worst-case complexity.

For distributivity, as long as the multiplication operator is itself commutative, so that the word problem for the 'coefficients' can be solved, the word problem for the combination can be decided with Buchberger-like algorithms [19].

Introducing negation, giving us the theory ACUN of Abelian groups, moves the bound from ESPACE to polynomial-time: Buchberger's algorithm reduces to finding the Hermite normal form of an integer matrix [8]. For the theory of vector spaces, Gaussian elimination can solve the word problem in (sub)cubic time.

6.3.3 EMTs and the word problem via critical pairs

So, for a theory T, we need to build in to our E-graph a system equivalent in strength to a word problem solver (even if, as for $T = \mathrm{A}$, that system is partial, i.e. only a semidecision procedure).

Resolving T-critical pairs is a natural way to integrate such a solver. Since E-graphs are built around creating a confluent ground rewriting system that resolves all ordinary critical pairs, we can build EMTs for a theory T using an algorithm that resolves T-critical pairs to create a confluent ground rewriting system on terms modulo T.

7

Prior work

7.1 EMTs: prior attempts

That *some* notion of EMT ought to exist has been suggested multiple times before [26, 14, 17]. The hope is to be able to use theory-specific knowledge to integrate certain identities directly into the procedures, rather than having to time-consumingly match, seed, and merge as directed by the identities. We will briefly go over the key intutition behind these proposals, and show that they fail to take into account the T-critical pair idea we have outlined previously (informally for AC in section 2.4, formally for AC in section 3.5, and for general T in section 6.3.3).

7.1.1 Intuition 1: Canonizers

Zucker [26] notes that we have *canonizers* for many theories, which provide a way to solve the problem of ground equality modulo theories. A canonizer for a theory T is just some function on terms c such that, if $t_1 =_T t_2$, then $c(t_1) = c(t_2)$: that is, it shifts the problem from equality modulo T to just ordinary term equality.

7.1.2 Intutition 2: Containers

The signature for an E-graph, Σ, is equivalent to a certain kind of *container*: one that has a number of 'shapes' (corresponding to the function symbols) and each shape has an ordered collection of slots (with length given by the arity of the shape). Thus, it is natural to consider generalizing this to containers of more general varieties. If one assigns to a single symbol one shape of every arity , with ordered slots, this gives the theory of AU: terms like $+()$, $+(a, b)$, $+(a, b, c)$ are all covered by this notion, and one doesn't need to arbitrarily split a given list into binary uses of $+(-, -)$ and nilary uses of e. If the ordering constraint is dropped on the collections of slots, we get multiset containers, suitable for ACU, and moving to set containers gives something suitable for ACIU – ACU with idempotence.

The second intuition also arises naturally out of concrete implementation work on E-graphs. Given a signature with an ACU binary operator f with unit e, and some other operators g, h, we can imagine building a parameterized data structure for term-fragments along the lines of the following pseudo-Haskell:

```
data TermF a
```

```
        = E
        | F a a
        | G a a a
        | H a
```

Looking at this structure and considering the ACU nature of f and e, it is only natural to change this to

```
data TermF a
        = Fs (Multiset a)
        | G a a a
        | H a
```

for an appropriate type constructor of multisets.

7.1.3 Canonizers and containers and the problem of flattening

The first intuition naturally connects to the second intuition if we consider the generalized idea of 'container' as representing some canonical form. Both approaches at first glance seem to take care of a lot of potential fiddling with identities in such theories. However, there is one issue: while we manage to handle some part of the theory with this approach, there are still two derived identities we need to handle.

For instance, consider an AU pair of operators $+, 0$. The identities are

$$a + (b + c) = (a + b) + c \tag{7.1}$$
$$a + 0 = a \tag{7.2}$$
$$0 + a = a \tag{7.3}$$

Using a list representation, these become

$$[a, [b, c]] = [[a, b], c] \tag{7.4}$$
$$[a, []] = a \tag{7.5}$$
$$[[], a] = a \tag{7.6}$$

These identities are not entirely automatic just by switching to lists: we need the additional embedding and and flattening laws

$$a = [a] \tag{7.7}$$
$$[[a_1, \ldots, a_n], [b_1, \ldots, b_m], \ldots] = [a_1, \ldots, a_n, b_1, \ldots, b_m, \ldots] \tag{7.8}$$

Or in other words, every term can be embedded in a list of just that term, and a list of lists can be flattened by concatenating all of its elements.[1]

Equation 7.7 and equation 7.8 together with the basic rules of lists are what is required to prove equation 7.4, equation 7.5, and equation 7.6.

[1] Readers inclined to abstract nonsense may recognize these equations as defining a monad for the algebraic theory of associativity [20])

It is the need to handle embedding and flattening that gives rise to problems with an approach based on just canonizers or containers. We will then also see that embedding and flattening are the two rules that govern the formation of T-critical pairs.

A typical example of where these approaches break down is given by our example from section 2.3:

$$a + b = b \tag{7.9}$$
$$a + a = c \tag{7.10}$$
$$c + b = d \tag{7.11}$$

The goal is to prove simply that $b = d$, which follows from the simple deduction

$$\begin{aligned} b \\ &= a + b \\ &= a + (a + b) \\ &= (a + a) + b \\ &= c + b \\ &= d \end{aligned}$$

If we try the containers approach, we start by building up an E-graph with the following E-classes: $e_1 = \{b, [a, b]\}, e_2 = \{c, [a, a]\}, e_3 = \{d, [c, b]\}$. However, this is by itself, entirely inert. We need to use flattening to reason that, since $b = [a, b]$, that $[a, b] = [a, [a, b]] = [a, a, b]$. Then we need to use the converse of flattening to notice that $[a, a, b] = [[a, a], b] = [c, b] = d$.

For the canonizer approach, we can easily canonize the input equations: indeed, all of them have both sides in canonical form already. What is missing, then? Well, if we view the E-graph as a rewrite system, there is of course an A-critical pair between the rewrite $a + b \to e_1$ and $a + a \to e_2$. The critical pair is $a + a + b$, which can be rewritten as either $a + e_1$ or $e_2 + b$.

We conclude that canonizers and containers are not enough: they can rewrite terms that might appear in equations to a canonical form, but this is not enough to make the equations $s_i = t_i$ when flattened and oriented into the form $s_i \to^* e_j \leftarrow^* t_i$ to naturally form a T-confluent system rewrite.

7.1.4 When canonizers and containers are enough

There are some cases where canonizers or containers avoid this problem: for instance, consider the theory C. A critical pair mod C can only be of the form $e_1 \leftarrow a + b =_C b + a \to e_2$. If we always canonize (mod C) both sides of equations when we orient them (say, by sorting according to some stable order), then we will have a confluent system mod C. (This requires some care to maintain the ordering given that a and b might be E-class ids and subject to change via union-find and rebuilding).

We have seen that C is a case of a theory that is *weakly term acyclic* (section 6.2.2).

Conjecture: all weakly term acyclic theories can be solved by a canonizer/container-style approach.

7.2 Congruence closure modulo theories

There has been work on congruence closure modulo various theories:

- [2] describe a congruence-closure modulo AC as building a rewriting system over an extended signature, and provide an algorithm for converting this back to a rewriting system over the original signature
- [19] provides an in-depth description of congruence-closure modulo AC and various other variants that can be covered by Buchberger's algorithm (polynomials, rings, etc.)
- [9] describes a congruence-closure modulo AC algorithm, and [10] extends this in a modular way to handle additional identities for identities including combinations of idempotency, nilpotency, units, and cancelativity
- [4] gives a congruence-closure modulo a 'solvable' theory. Unlike various modulo-AC variants, this requires the ability to, for instance, 'solve' the equation $3x = 2y$ by turning it into $x = \frac{2}{3}y$, which requires inverses for the relevant operators.

8

Conclusion and future work

EMTs provide a good basis for integrating certain common well-behaved theories more tightly into the core of E-graphs, rather than allowing the potentially explosive equality saturation to handle them. However, there are still fundamental limitations to how 'nice' EMTs can be: they expand the E-graph during the resolution of critical pairs, unlike congruence-closure which can only shrink it, and the possible performance gains are bounded tightly by the complexity of solving the word problems for the relevant theories.

8.1 Future work

8.1.1 EMTs for other theories

Given that there are already several variants of congruence-closure modulo theories (Section 7.2), it would be useful to evaluate EMTs on these theories, including implementing and evaluating E-matching modulo them. Many of them require Buchberger-like algorithms similar to the one we have presented for AC. Variations of this algorithm for certain theories (Section 6.3.2) can be polynomial-time rather than ESpace.

8.1.2 E-matching and EMTs

Our reduction of E-matching to a kind of E-analysis in 4.3.1 doesn't take into account the literature on relational E-matching [25]. Whether the variety of strategies for relational E-matching can be reduced in a similar way remains to be seen.

8.1.3 The three Es seen through rewriting

Our brief sketch in section 4.4.3.1 of an E-matching system presented as a critical-pair-based procedure still lacks some details, and it remains to be proven that it can always generate the a terminating set of rewrites for the potentially infinite sets that can occur in AC-matches.

8.1.4 Improving AC-closure

Buchberger's algorithm has the F_4 and F_5 variants and an extensive literature on optimizing it. Importing these variants into an EMT implementation could provide significant speedup. Also, as discussed in Section 5.2.4, there is the potential to make the AC-closure more incremental and reduce the cost associated with computing it.

Given that E-graphs work over an extended signature, is it worthwhile to compress the AC-rules by shrinking the signature? For instance, if we have AC-rules $e_1 + e_2 \to e_3$ and $e_3 + e_4 \to e_5$, and e_3 doesn't appear among the ordinary E-node rules, then we could shrink the system to just $e_1 + e_2 + e_4 \to e_5$ and delete e_3.

8.1.5 Constrained matching

As mentioned in Section 4.4.3, adding constraints to patterns for matching could significantly cut down the search space for matching modulo theories when the full space of matches, which includes a lot of redundant symmetries, is not needed.

Similar ideas to prune large search spaces are common in automatic theorem proving. See, for instance, [15, Chapter 7, Section 5].

8.1.6 Realistic applications

While EMT(AC)s may be fast*er* than E-graphs, it remains to be seen if they are fast *enough* to make significant progress in various application domains that have struggled with the explosive nature of AC-saturation.

Bibliography

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998. URL: https://books.google.com/books?hl=sv&lr=&id=N7BvXVUCQk8C&oi=fnd&pg=PR9&dq=baader+nipkow&ots=cQ-4LiZxZh&sig=3IGr91DGoASOjhIJpb_mHnwqtnY (visited on 08/04/2025).

[2] L. Bachmair et al. "Congruence Closure Modulo Associativity and Commutativity". In: *Frontiers of Combining Systems*. Ed. by Hélène Kirchner and Christophe Ringeissen. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 1794. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 245–259. ISBN: 978-3-540-67281-4. DOI: 10.1007/10720084_16. URL: http://link.springer.com/10.1007/10720084_16 (visited on 02/24/2026).

[3] Leo Bachmair and Ashish Tiwari. "Abstract Congruence Closure and Specializations". In: *Automated Deduction - CADE-17*. Ed. by David McAllester. Berlin, Heidelberg: Springer, 2000, pp. 64–78. ISBN: 978-3-540-45101-3. DOI: 10.1007/10721959_5.

[4] Sylvain Conchon et al. "CC(X): Semantic Combination of Congruence Closure with Solvable Theories". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007) 198.2 (May 6, 2008), pp. 51–69. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.080. URL: https://www.sciencedirect.com/science/article/pii/S1571066108002958 (visited on 02/08/2026).

[5] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-16720-6. DOI: 10.1007/978-3-319-16721-3. URL: https://link.springer.com/10.1007/978-3-319-16721-3 (visited on 03/17/2026).

[6] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. "Variations on the Common Subexpression Problem". In: *Journal of the ACM* 27.4 (Oct. 1980), pp. 758–771. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322217.322228. URL: https://dl.acm.org/doi/10.1145/322217.322228 (visited on 02/08/2026).

[7] Guy Frankel et al. "Syntax-guided synthesis with counterexample-guided E-graphs: A work-in-progress report". In: *The 23rd International Workshop on Satisfiability Modulo Theories*. 2025. URL: https://www.research.ed.ac.uk/en/publications/syntax-guided-synthesis-with-counterexample-guided-e-graphs-a-wor/ (visited on 06/20/2026).

[8] Abdelilah Kandri-Rody, Deepak Kapur, and Paliath Narendran. "An ideal-theoretic approach to word problems and unification problems over finitely presented commutative algebras". In: *Rewriting Techniques and Applications*. Ed. by Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer, 1985, pp. 345–364. ISBN: 978-3-540-39679-6. DOI: 10.1007/3-540-15976-2_17.

[9] Deepak Kapur. "A Modular Associative Commutative (AC) Congruence Closure Algorithm". In: *LIPIcs, Volume 195, FSCD 2021* 195 (2021). Ed. by Naoki Kobayashi. Artwork Size: 21 pages, 711938 bytes ISBN: 9783959771917 Medium: application/pdf, 15:1–15:21. ISSN: 1868-8969. DOI: 10.4230/LIPICS.FSCD.2021.15. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2021.15 (visited on 03/17/2026).

[10] Deepak Kapur. "Modularity and Combination of Associative Commutative Congruence Closure Algorithms enriched with Semantic Properties". In: *Logical Methods in Computer Science* Volume 19, Issue 1 (Mar. 14, 2023), p. 8693. ISSN: 1860-5974. DOI: 10.46298/lmcs-19(1:19)2023. URL: https://lmcs.episciences.org/8693 (visited on 02/24/2026).

[11] Ernst W Mayr and Albert R Meyer. "The complexity of the word problems for commutative semigroups and polynomial ideals". In: *Advances in Mathematics* 46.3 (Dec. 1, 1982), pp. 305–329. ISSN: 0001-8708. DOI: 10.1016/0001-8708(82)90048-2. URL: https://www.sciencedirect.com/science/article/pii/0001870882900482 (visited on 03/11/2026).

[12] Greg Nelson and Derek C. Oppen. "Fast Decision Procedures Based on Congruence Closure". In: *Journal of the ACM* 27.2 (Apr. 1980), pp. 356–364. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322186.322198. URL: https://dl.acm.org/doi/10.1145/322186.322198 (visited on 02/08/2026).

[13] Carl-Fredrik Nyberg-Brodda. *G. S. Tseytin's seven-relation semigroup with undecidable word problem*. Jan. 22, 2024. DOI: 10.48550/arXiv.2401.11757. arXiv: 2401.11757[math]. URL: http://arxiv.org/abs/2401.11757 (visited on 02/27/2026).

[14] Pavel Panchekha. *LIN-graphs, an Egraph modulo T*. Pavel's Blog. May 23, 2025. URL: https://pavpanchekha.com/blog/lin-graphs.html.

[15] Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*. Vol. 1. Elsevier, 2001. URL: https://books.google.com/books?hl=sv&lr=&id=HxaWA4lep_kC&oi=fnd&pg=PP1&dq=handbook+of+automated+reasoning&ots=SOYBCqhWIi&sig=VvmK6rsed81TCue3S0zpLynCG7o (visited on 02/04/2026).

[16] Tarik Rosin, Marcel Ullrich, and Sebastian Hack. *Associativity and Commutativity in Equality Saturation*. 2026. URL: https://compilers.cs.uni-saarland.de/papers/ullrich_egraphs2026-paper17.pdf (visited on 06/19/2026).

[17] Saul Shanabrook. *Custom Data Structures in E-Graphs*. PLSE Blog. Feb. 24, 2026. URL: https://uwplse.org/2026/02/24/egglog-containers.html.

[18] Dan Suciu, Yisu Remy Wang, and Yihong Zhang. *Semantic foundations of equality saturation*. Jan. 5, 2025. DOI: 10.48550/arXiv.2501.02413. arXiv: 2501.02413[cs]. URL: http://arxiv.org/abs/2501.02413 (visited on 02/04/2026).

[19] Ashish Tiwari. *Decision procedures in automated deduction*. State University of New York at Stony Brook, 2000. URL: https://search.proquest.com/openview/30046982652b13cadc7b85d653cdf547/1?pq-origsite=gscholar&cbl=18750&diss=y (visited on 03/18/2026).

[20] Anthony Voutas. "The basic theory of monads and their connection to universal algebra". In: (2012). URL: https://ncatlab.org/nlab/files/Voutas-Monads.pdf (visited on 06/25/2026).

[21] Yisu Remy Wang, Max Willsey, and Dan Suciu. "Free Join: Unifying Worst-Case Optimal and Traditional Joins". In: *Proceedings of the ACM on Management of Data* 1.2 (June 13, 2023), pp. 1–23. ISSN: 2836-6573. DOI: 10.1145/3589295. URL: https://dl.acm.org/doi/10.1145/3589295 (visited on 04/14/2025).

[22] Max Willsey et al. *Egg > BackoffScheduler*. Version 0.11.0. 2026. URL: https://github.com/egraphs-good/egg/blob/v0.11.0/src/run.rs#L816.

[23] Max Willsey et al. "egg: Fast and extensible equality saturation". In: *Proceedings of the ACM on Programming Languages* 5 (POPL Jan. 4, 2021), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3434304. URL: https://dl.acm.org/doi/10.1145/3434304 (visited on 02/08/2026).

[24] Youwei Xiao, Yuyang Zou, and Yun Liang. *SkyEgg: Joint Implementation Selection and Scheduling for Hardware Synthesis using E-graphs*. Nov. 19, 2025. DOI: 10.48550/arXiv.2511.15323. arXiv: 2511.15323[cs.PL]. URL: http://arxiv.org/abs/2511.15323 (visited on 06/20/2026).

[25] Yihong Zhang et al. "Relational E-matching". In: *Proceedings of the ACM on Programming Languages* 6 (POPL Jan. 16, 2022), pp. 1–22. ISSN: 2475-1421. DOI: 10.1145/3498696. URL: https://dl.acm.org/doi/10.1145/3498696 (visited on 04/14/2025).

[26] Philip Zucker. *Omelets Need Onions: E-graphs Modulo Theories via Bottom-up E-matching*. Apr. 19, 2025. DOI: 10.48550/arXiv.2504.14340. arXiv: 2504.14340[cs]. URL: http://arxiv.org/abs/2504.14340 (visited on 01/20/2026).