15-414: Bug Catching: Automated Program Verification

Lecture Notes on
Deciding Uninterpreted Functions (and Arrays)

Matt Fredrikson

Carnegie Mellon University

Lecture 19
Thursday, March 30, 2023

1 Introduction

In the previous lectures we have studied decision procedures for propositional logic. However, verification conditions that arise in practice often refer to objects from infinite domains, and cannot be reduced to propositional satisfiability. For example, the following formula talks about linear arithmetic over real numbers as well as function application:

$$(x_2 \geq x_1) \land (x_1 - x_3 \geq 2) \land (x_3 \geq 0) \land f(f(x_1) - f(x_2)) \neq f(x_3)$$

Today we will discuss the problem of deciding *satisfiability modulo theories*, and take a closer look at specialized procedures for two first-order theories: arrays, and equality with uninterpreted functions. We will start by considering formulas from just one theory, and return to the question of incorporating multiple theories in a later lecture.

Learning Goals

- A procedure for deciding the satisfiability of formulas in the conjunctive quantifier-free fragment of the theory of arrays.
- The theory of *equality and uninterpreted functions* (EUF), and how it is used to partially decide the theory of arrays.
- The *congruence closure* algorithm for deciding conjunctive, quantifier free formulas in EUF.

2 First-order theories

A first-order theory T is defined by the following components.

- It's signature Σ is a set of constant, function, and predicate symbols.
- It's set of axioms $\mathcal{A}$ is a set of closed first-order logic formulae in which only constant, function, and predicate symbols of Σ appear.

Having defined a theory's signature and axioms, we can reason about the same type of properties related to the semantics of a formula as we have been so far, namely validity and satisfiability.

Definition 1 (T-valid)**.** A Σ-formula φ is valid in the theory T (T-valid), if *every model* M that satisfies the axioms of T (i.e., $I \models A$ for every $A \in \mathcal{A}$) also satisfies φ (i.e., $I \models \varphi$).

Definition 2 (T-satisfiable)**.** Let T be a Σ-theory. A Σ-formula φ is T-satisfiable if there *exists a model* M such that $I \models A$ and $I \models \varphi$.

Definition 3 (T-decidable)**.** A theory T is decidable if $T \models \varphi$ is decidable for every Σ-formula. That is, there exists an algorithm that always terminate with "yes" if φ is T-valid or with "no" if φ is T-invalid.

2.1 Examples

We have already discussed several first-order theories in this class, including the theory of arrays (Lecture 8) and the theory of bit vector arithmetic (Lecture 16). We will dive into the theory of arrays in more detail in the next section, but first it may be helpful to illustrate a few examples of first-order theories.

Bit vector arithmetic. The theory of ℓ-width bit vector arithmetic has constant symbols corresponding to all possible sequences of ℓ binary $\{0, 1\}$ values. The function symbols are the usual arithmetic operations $(+, -, *, /, \ldots)$ and bitwise operations $(\sim, \&, |, \oplus, \ll, \gg)$. We did not cover the axioms explicitly, but they correspond to the semantics that we studied: arithmetic modulo 2^n, as well as those relating the effect of bitwise operations.

Presburger arithmetic. The theory of Presburger arithmetic has just two constants: 0 and 1. The only function is addition, and the only predicate is equality. We will not enumerate all of the axioms, but they suffice to give meaning to these symbols. For example, the axiom $x+0 = x$ identifies 0 as the identity element of addition. The axioms of Presburger arithmetic are not a finite set, as illustrated by the induction "schema":

$$f(0) \land (\forall x. f(x) \rightarrow f(x+1)) \rightarrow \forall x. f(x)$$

Here, f is not a constant from the signature, but a placeholder for an arbitrary formula. There is one such axiom for each formula, making the cardinality of the axioms infinite.

3 Theory of Arrays

Recall from Lecture 8 that in order to reason about the mutable arrays used in many of our programs, we formalized a more abstract "logical" theory of arrays that reflected the essential properties we needed. We assumed that arrays are indexed by some type ι, that their elements belonged to a separate type τ We defined two operations, read and write.

$\mathsf{read}\,a\,i : \tau$	read from array a at index i
$\mathsf{write}\,a\,i\,v : \mathsf{array}\,\iota\,\tau$	write to array a at index i

Importantly, write does not mutate a, but we instead think of it as a constructor that denotes a new array distinct from a, obtained by writing the value v to index i of a, and keeping the values at other indices unchanged.

There are two axioms that define the meaning of write terms, called the *read-over-write* axioms.

$$\begin{aligned} i = k &\rightarrow \mathsf{read}\,(\mathsf{write}\,a\,i\,v)\,k = v \\ i \neq k &\rightarrow \mathsf{read}\,(\mathsf{write}\,a\,i\,v)\,k = \mathsf{read}\,a\,k \end{aligned}$$

Recall that we do not worry about whether array accesses are "in bounds", and assume that there are no bounds for these abstract arrays. When using the theory to reason about *real* arrays, additional constraints on the indices are used to encode the bounds.

To reason about the equality of the arrays themselves, i.e. as in the case of a formula like the one shown in (1), we adopted the *extensionality* axiom.

$$\mathsf{write}\,a\,i\,(\mathsf{read}\,a\,i) \stackrel{?}{=} a \tag{1}$$

Extensionality just says that two arrays are equal whenever their values at all indices are equal.

$$(\forall i.\, \mathsf{read}\,a\,i = \mathsf{read}\,a'\,i) \rightarrow a = a'$$

In today's lecture, we will adopt another axiom for arrays called *congruence*:

$$i = j \rightarrow \mathsf{read}\,a\,i = \mathsf{read}\,a\,j$$

While the congruence axiom may seem like a basic consequence of equality, it plays a central role in the procedure that we will use to decide the satisfiability of array formulas. For reasons that will become clear shortly, we also make the axioms of equality explicit. Namely, the predicate $=$ satisfies the axioms of reflexivity, symmetry, and transitivity.

$$\begin{aligned} &x = x && \text{(reflexive)} \\ &x = y \rightarrow y = x && \text{(symmetric)} \\ &x = y \wedge y = z \rightarrow x = z && \text{(transitive)} \end{aligned}$$

We now turn to a procedure for deciding formulas in the theory of arrays.

3.1 A Partial Decision Procedure

For now, we will restrict ourselves to considering a *fragment* of the theory of arrays, and thus we will only see how to decide whether array formulas subject to syntactic conditions are satisfiable. We will consider quantifier-free formulas whose logical connectives are only conjunction and negation. Further, the negations are allowed to occur only over atoms, and not other conjunctions or negations. So for example, the following formula is in the quantifier-free conjunctive fragment:

$$i = j \land i \neq k \land \mathsf{read}\, a\, j = w \land \neg(\mathsf{read}\, (\mathsf{write}\, a\, i\, v)\, j = \mathsf{read}\, a\, j)$$

But the following one is not, because negation occurs over a conjunction:

$$i = j \land i \neq k \land \neg(\mathsf{read}\, a\, j = w \land \mathsf{read}\, (\mathsf{write}\, a\, i\, v)\, j = \mathsf{read}\, a\, j)$$

This may seem like a limitation at first, but in future lectures we will see that it is not.

The procedure for the conjunctive, quantifier-free fragment proceeds by first removing all of the read-over-write terms in the formula using the axioms, and then reasons about the resulting formula, which does not contain any write terms, by systematically reasoning about the remaining equalities involving read terms.

Consider the formula shown in (2).

$$x \neq y \land \mathsf{read}\, a\, i = x \land \mathsf{read}\, a\, j = y \land \mathsf{read}\, (\mathsf{write}\, a\, i\, (\mathsf{read}\, a\, j))\, i = \mathsf{read}\, (\mathsf{write}\, a\, j\, x)\, i \quad (2)$$

We will now illustrate the steps taken by the procedure to decide whether a satisfying assignment exists for the formula.

Step 1: remove the write terms. The procedure will select one of the two read over write terms appearing in the formula.

1. $\mathsf{read}\, (\mathsf{write}\, a\, i\, (\mathsf{read}\, a\, j))$
2. $\mathsf{read}\, (\mathsf{write}\, a\, j\, x)\, i$

Starting with the first, our goal is to remove the write term appearing in the formula. In order to preserve the meaning of the original formula, the read-over-write axioms tell us that there are two cases to consider: either the index being accessed by the outermost read is equal to the one being modified by the write, or it is not. In the first case, the corresponding axiom would tell us that we can replace the entire read-over-write term with the value written, so it would become $\mathsf{read}\, a\, j$. We account for the fact that this replacement is only correct when the array indices match by conjoining the appropriate formula (i.e., $i = i$). Although it is clear that conjoining $i = i$ will change nothing, we do it anyway to make the steps taken by the procedure explicit. We arrive at the formula:

$$x \neq y \land \mathsf{read}\, a\, i = x \land \mathsf{read}\, a\, j = y \land \mathsf{read}\, a\, j = \mathsf{read}\, (\mathsf{write}\, a\, j\, x)\, i \land i = i$$

We still must consider the case where the array accessed by the read is different from the one modified by the write. While we know that this case can never happen, as it would imply that $i \neq i$, we will write the formula down to illustrate the application of the other read-over-write axiom, which has us replace the write inside the read with the original array being written to.

$$x \neq y \land \mathsf{read}\, a\, i = x \land \mathsf{read}\, a\, j = y \land \mathsf{read}\, a\, i = \mathsf{read}\, (\mathsf{write}\, a\, j\, x)\, i \land i \neq i$$

In the formulas resulting from both of the cases that we have considered so far, there are still write terms remaining. However, it is straightforward that we do not need to proceed further with the second formula, which has $i \neq i$ conjoined, because this is an immediate contradiction that makes the formula equivalent to *false*.

The procedure continues in this fashion on the formulas for any case that still contains a write, splitting cases and replacing read-over-write instances as directed by the axioms. In the first formula, the cases split on $i = j$ and $i \neq j$, and lead to the following two formulas:

$$\begin{array}{l} x \neq y \land \mathsf{read}\, a\, i = x \land \mathsf{read}\, a\, j = y \land \mathsf{read}\, a\, j = x \land i = i \land i = j \\ x \neq y \land \mathsf{read}\, a\, i = x \land \mathsf{read}\, a\, j = y \land \mathsf{read}\, a\, j = \mathsf{read}\, a\, i \land i = i \land i \neq j \end{array} \tag{3}$$

At this point there are no further write terms in either formula, so the procedure continues by reasoning about the equalities over read terms.

Step 2: decide the satisfiability of equalities over read terms. Looking at the formulas in (3), we can reason intuitively that they are not satisfiable. For the top formula, the constraint $i = j$ would imply, via the congruence axiom, that $\mathsf{read}\, a\, i = x = \mathsf{read}\, a\, j = y$. But the formula also has that $x \neq y$, so this is not satisfiable. For the second formula, recalled that $i \neq j$ implied that $\mathsf{read}\, a\, j = \mathsf{read}\, a\, i$. But $\mathsf{read}\, a\, i = x$ and $\mathsf{read}\, a\, j = y$, and $x \neq y$, so this also cannot be satisfied.

Notice that this reasoning only depends on what we know about equality and congruence. We essentially treat each array as an aribtrary function, and the read operation as the application of that function at the index. So if we assume that f_a is a function corresponding to the array a, we might as well have written the formulas in (3) in terms of f_a and its application rather than read.

$$\begin{array}{l} x \neq y \land f_a(i) = x \land f_a(j) = y \land f_a(j) = x \land i = i \land i = j \\ x \neq y \land f_a(i) = x \land f_a(j) = y \land f_a(j) = f_a(i) \land i = i \land i \neq j \end{array} \tag{4}$$

Thus, the second step of our procedure will construct a function symbol for each array in the formula, replace all of the read terms in the formula with function application. The result can then be solved using a procedure for the *theory of equality with uninterpreted functions*, which is the subject of the next section.

4 Theory of Equality with Uninterpreted Functions (EUF)

The **theory of equality with uninterpreted functions** T_{E} has a signature that consists of a single binary predicate $=$, and all possible constant $(a, b, c, x, y, z, \ldots)$ and function

$(f, g, h, \ldots)$ symbols:

$$\Sigma_{\mathsf{E}} : \{=, a, b, c, \ldots, f, g, h, \ldots\}$$

The axioms of T_{E} define the usual meaning of equality (reflexivity, symmetry, and transitivity), as well as *functional congruence*.

1. $\forall x . x = x$ (reflexivity)
2. $\forall x, y . x = y \to y = x$ (symmetry)
3. $\forall x, y, z . x = y \land y = z \to x = z$ (transitivity)
4. $\forall x, y . x = y \to f(\bar{x}) = f(\bar{y})$ (congruence)

Function congruence states that whenever the arguments to a function are equal, then the function's value at those arguments must be equal as well. Observe that this is equivalent to the congruence axiom from the theory of arrays, if we replace function applications with their corresponding read terms. In fact, it generalizes array congruence, because it can also stated so that it applies to functions with multiple arguments. If $\bar{x}$ and $\bar{y}$ are sequences $x_0, \ldots, x_n$ and $y_0, \ldots, y_n$ of variables, then Equation 5 formalizes congruence over n-ary functions.

$$\forall \bar{x}, \bar{y} . (\textstyle\bigwedge_{i=1}^{n} x_i = y_i) \to f(\bar{x}) = f(\bar{y}) \tag{5}$$

For the purposes of today's lecture, we only need to consider unary functions of a single argument, because they are sufficient to capture the meaning of read terms in the theory of arrays.

We note that many treatments of this theory also include predicate symbols, and have a corresponding notion of predicate congruence. The algorithm that we present later will work on formulas that include predicates as well, with minimal modifications; our use of EUF to reason about arrays does not require them, so we leave predicates out of the theory for the rest of the lecture.

Example 4. Consider the Σ-formula φ

$$f(f(f(a))) = a \land f(f(f(f(f(a))))) = a \land f(a) \neq a$$

φ is T_{E}-unsatisfiable. We can make the following intuitive argument: substituting a for $f(f(f(a)))$ in $f(f(f(f(f(a))))) = a$ by the first equality yields $f(f(a)) = a$; substituting a for $f(f(a))$ in $f(f(f(a))) = a$ according to this new equality yields $f(a) = a$, contradicting the literal $f(a) \neq a$. Formally, we can apply the axioms of T_{E} and derive the same contradiction:

1. $f(f(f(f(a)))) = f(a)$ first literal of φ (congruence)
2. $f(f(f(f(f(f(a)))))) = f(f(a))$ step 1 (congruence)
3. $f(f(a)) = f(f(f(f(f(f(a))))))$ step 2 (symmetry)
4. $f(f(a)) = a$ step 3 and second literal of φ (transitivity)

4.1 Deciding EUF: The Congruence Closure Algorithm

Each positive positive literal $s = t$ of a Σ-formula φ over T_{E} asserts an equality between two terms s and t. Applying the axioms of T_{E} produces more equalities over terms that occur in φ. Since there are only a finite number of terms in φ, only a finite number of equalities among these terms are possible. Hence, one of two situations eventually occurs: either some equality is formed that directly contradicts a negative literal $s' \neq t'$ of φ; or the propagation of equalities ends without finding a contradiction. These cases correspond to T_{E}-unsatisfiability and T_{E}-satisfiability, respectively, of φ. In this section, we will formally describe this procedure as forming the **congruence closure** of the equality relation over terms asserted by φ.

Models of equality. We begin by introducing the notion of a *congruence relation* in Definition 5.

Definition 5 (Congruence relation, congruence class)**.** Consider a set S and functions $F = \{f_1, \ldots, f_n\}$. A relation R over S is a *congruence relation* if for every function $f \in F$, it satisfies the following:

1. Reflexive: $\forall_s \in S. s \, R \, s$
2. Symmetric: $\forall_{s_1,s_2} \in S. s_1 \, R \, s_2 \rightarrow s_2 \, R \, s_1$
3. Transitive: $\forall_{s_1,s_2,s_3} \in S. s_1 \, R \, s_2 \wedge s_2 \, R \, s_3 \rightarrow s_1 \, R \, s_3$
4. Congruent: $\forall_{s,t} s \, R \, t \rightarrow f(s) \, R \, f(t)$

We say that two elements $x, y \in S$ are in the same *congruence class* of R whenever $x \, R \, y$, and write $[x]_R$ to denote the set of elements in x's congruence class.

You may have noticed that the requirements of a congruence relation mirror the axioms of our present theory. Suppose that we are shown a congruence relation R over the set $S = \{a, b, f(a), f(b)\}$. The properties of congruence relations make it possible for us to construct a satisfiable EUF formula from R. For example, if R relates the pairs $\{(a, b), (f(a), f(b))\}$[1] from S, then we could derive:

$$a = b \wedge f(a) = f(b) \wedge a \neq f(a) \wedge a \neq f(b) \wedge b \neq f(a) \wedge b \neq f(b)$$

In other words, any pair related by R appears in an equality literal, and any possible pair *not* in R in a negative equality literal. We know that this formula will be satisfiable, because everything that is equated came from R, which is reflexive, symmetric, transitive, and congruent.

If we could "reverse" this reasoning, and derive a congruence relation for a given formula, then perhaps we could decide that the formula is satisfiable. For example, given the formula $P \equiv a = f(x) \wedge a = g(y) \wedge x \neq y$, then $R = \{(a, (f(x)), (a, g(y)), \ldots)\}$

---

[1] We do not include the symmetric pairs (b, a) and $(f(a), f(b))$ explicitly to save space, but they must be in R for it to be a congruence relation.

would be such a relation. Note that the ellipses refer to an infinite set of pairs that follow from nested applications of f and g via congruence. For example, because $a\,R\,f(x)$, congruence says that $f(a)\,R\,f(f(x))$, and that $f(f(a))\,R\,f(f(f(x)))$, and $\ldots$, must also be true. Congruence relations will always be impossible to write down for this reason, and we will instead use the convention of denoting them by their congruence classes only over the terms that appear in the formula. We would thus denote R in this way as $\{\{a, f(x), g(y)\}, \{x\}, \{y\}\}$.

We can say that R *models* P, written $R \models P$, as it demonstrates the satisfiability of P. In whatever domain the terms of P range over, we could assign a unique element for each congruence class of R. Then any assignment where variables and function applications map to the element for their congruence class will satisfy P.

To see this concretely, let us assume that a, b, x, y, f, and g range over integers. The current relation R has three equivalence classes: one containing $a, f(x), g(y)$, another containing x, and one containing y. If we let 0 be the element for the first class, 1 be for the second, and 2 for the third, then a satisfying assignment M would be:

$$M(a) = 0, M(x) = 1, M(y) = 2, M(f) = M(g) = [0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0]$$

To conclude, given a congruence relation over the terms appearing in a formula, we can construct an assignment to the variables and function values appearing in that formula. Moreover, this assignment will be consistent with the axioms of equality, as well as with function congruence.

Minimal models. Observe that not all congruence relations over $\{a, f(x), g(y), x, y\}$ from the example in the previous paragraph work as models of P. For any set S of terms in a formula P, the relation containing one congruence class is always trivially a congruence relation. This corresponds to the maximal congruence relation $R^{\max}$ over S, and if there is a negative equality literal in P, then $R^{\max}$ will not model P. In the example from the previous paragraph, this relation would allow x and y to be assigned to the same element 0, because $x\,R^{\max}\,y$.

In general, a congruence relation R does not model a formula P whenever there exist a set of terms s, t where $s\,R\,t$ and a negative equality $s \neq t$ in P. Thus, when searching for a relation that models a formula, we want to find the *minimal* congruence relation in order to avoid relating terms that conflict with a negative equality in P. This motivates the definition of *congruence closure*, detailed in Definition 6.

Definition 6 (Congruence closure)**.** The *congruence closure* R^{cong} of the binary relation R over S is the unique relation which satisfies:

- R^{cong} relates everything that R does: $R \subseteq R^{\text{cong}}$.
- R^{cong} is the smallest congruence relation satisfying (1). If R' is a congruence relation that satisfies (1), then $R^{\text{cong}} \subseteq R'$.

Note that the congruence closure of a given relation always exists, because $R^{\max}$ is a congruence relation; in the "worst" case, it may also be the smallest congruence relation containing R.

A bit of thought should convince you that if we begin with a relation R that captures the equality literals in P, and compute its congruence closure, then whenever P is satisfiable, R^{cong} will model it. Returning to the previous example,

$$P \equiv a = f(x) \land a = g(y) \land x \neq y$$

The relation that captures the equality literals in P is given by $R = \{(a, f(x)), (a, g(y))\}$ (omitting the necessary reflexive and symmetric pairs for clarity). The congruence closure of R is,

$$R = \{(a, f(x)), (a, g(y)), (f(x), g(y)), (f(a), f(f(x))), (g(a), g(g(y)), (f(f(x)), f(g(y))), \ldots\}$$

For both relations, the congruence classes restricted to $\{a, x, y, f(x), f(y)\}$ (i.e., the terms appearing in the formula), are $\{\{a, f(x), f(y)\}, \{x\}, \{y\}\}$. In other words, in this case we can find a model of P just by processing the equality literals that appear in it because the classes of R are identical to those of R^{cong}.

As you might expect, this isn't always the case. Consider the example from earlier in the notes.

$$\varphi : f(f(f(a))) = a \land f(f(f(f(f(a))))) = a \land f(a) \neq a$$

The initial relation is $R = \{(f^3(a), a), (f^5(a), a)\}$. The set of terms appearing in the formula are $S = \{a, f(a), f^2(a), f^3(a), f^4(a), f^5(a)\}$, so the initial relation gives classes $\{\{a, f^3(a), f^5(a)\}, \{f(a)\}, \{f^2(a)\}, \{f^4(a)\}\}$. If we assign, for example, $f(a)$ and $f^4(a)$ to different elements, then congruence is violated because $a \, R \, f^3(a)$. So in this case we do in fact need to compute the congruence closure, which has just one class:

$$\{\{a, f(a), f^2(a), f^3(a), f^4(a), f^5(a)\}\}$$

In other words, everything in the formula must be related.

In this case, the congruence closure conflicts with $f(a) \neq a$. Can we conclude that the formula is not satisfiable based on just that evidence? Thanks to the work of Shostak [?] in the 1970's, we can answer this question affirmatively. The proof of this result is beyond the scope of today's lecture, but your intuition should serve you well in believing the claim. If the *minimal* relation that satisfies the reflexive, symmetric, transitive, and congruence axioms also conflicts with a negative equality in P, then how could one ever find a way to assign these terms to values that did not contradict the negative equality literal? Any such "satisfying" assignment would fail to account for one of the axioms, and thus fail as a model of the EUF formula.

The algorithm. To summarize what we have learned so far, we have that a congruence relation over the terms of a formula corresponds to an assignment that is consistent with the axioms of EUF. For a given relation, the congruence closure is the smallest congruence relation that contains R. If we begin with a relation that reflects the positive equality literals in a formula, and find its congruence closure, then the result will also give us a satisfying assignment if one exists.

We now turn to computing the congruence closure. The algorithm works explicitly with a representation of the congruence classes, rather than the relation itself. In the following, we will use the infix operator $\cong$ to refer to the congruence closure that is computed by the algorithm, and P to the formula being processed.

1. Let S_P be the set of all terms, and their subterms (recursively), in P.
2. Initialize $\cong$ by placing each element of S_P in its own congruence class.
3. For every positive literal $s = t$ in P, merge the congruence classes of s and t.
4. While $\cong$ changes, repeat the following:
   a) Propagate the congruence axiom, to account for any merged congruence classes from the previous step. For any $s \cong t$, if $f(\dots, s, \dots)$ and $f(\dots, t, \dots)$ are currently in different congruence classes, then merge them.
5. Check the negative equality literals in P against the computed $\cong$.
   - For any $s \neq t$ appearing in P, if $s \cong t$, then return that P is unsat.
   - Otherwise, $s \ncong t$ for all $s \neq t$ appearing in P, so return that P is sat.

Recall the assumptions that we have made about the formula P: it is in the conjunctive, quantifier-free frament of EUF. This is why it is possible to return unsat after finding just a single conflict with a negative equality literal. If there were a disjunction in P, then this conclusion would not be possible. For the conjunctive quantifier-free fragment, the algorithm is sound and complete.

Soundness means that whenever this procedure terminates, it produces the correct answer, and as we discussed earlier, Shostak [**?**] proved this. It is also complete, which means that it will always terminate, because the cardinality of the initial set of congruence classes is finite: each time a pair of congruence classes is merged, the procedure makes progress towards termination, which at the very least must occur when there is only one congruence class under $\cong$.

To efficiently implement the procedure, a popular approach is to take advantage of a union-find data structure. This is an acyclic graphical data structure where each node represents a term in S_P. Directed edges encode the subterm structure of P, i.e., the node for term $f(a)$ would have an edge to the node representing a. Congruence classes are also represented by directed edges, by arbitrarily picking a representative element from each congruence class, and drawing edges towards its node from all other members of its class.

Bradley and Manna describe such an implementation strategy [**?**, Chapter 9] that yields $O(e^2)$ runtime, where e is the number of positive equality literals in P, with $O(|S_P|)$ merge operations. Downey, Sethi, and Tarjan gave an algorithm with better average-case complexity, $O(e \log e)$ and $O(|S_P|)$ merges [**?**].

We'll conclude this section with a few examples to illustrate the procedure.

Example 7. Consider the formula:

$$P : f(a,b) = a \land f(f(a,b),b) \neq a$$

The subterm set S_P is $\{a, b, f(a,b), f(f(a,b),b)\}$, so we construct the initial relation by giving each element its own congruence class:

$$\cong_0 \colon \{\{a\},\{b\},\{f(a,b)\},\{f(f(a,b),b)\}\}$$

There is one equality in P, $f(a,b) = a$, so we merge the first and third congruence classes:

$$\cong_1 \colon \{\{a, f(a,b)\},\{b\},\{f(f(a,b),b)\}\}$$

Now we must check to see if there are congruences to propagate. Now that a and $f(a,b)$ are in the same class, we must determine whether any applications of $f(\cdot,b)$ to either of these terms resides in a different class. We see that $f(\cdot,b)$ applied to a, i.e. $f(a,b)$, is in a different class than $f(\cdot,b)$ applied to $f(a,b)$, i.e., $f(f(a,b),b)$. So we merge them, giving the relation:

$$\cong_2 \colon \{\{a, f(a,b), f(f(a,b),b)\},\{b\}\}.$$

As there are no further applications of f in any but the first equivalence class, there are no further opportunities to propagate congruence, so $\cong_2$ is the congruence closure of $\cong_0$. The last step of the procedure scans the negative literals in P to determine whether $\cong_2$ is a model. In this case, it is not, because there is one negative literal, $f(f(a,b),b) \neq a$, but these terms are in the same congruence class of $\cong_2$. Thus, the formula is `unsat`.

Example 8. Now we'll return to the example from earlier, but derive the congruence closure via the algorithm.

$$P : f(f(f(a))) = a \land f(f(f(f(f(a))))) = a \land f(a) \neq a$$

As we said before, the subterm set is $S_P = \{a, f(a), f^2(a), f^3(a), f^4(a), f^5(a)\}$, so the initial relation is:

$$\cong_0 \colon \{\{a\},\{f(a)\},\{f^2(a)\},\{f^3(a)\},\{f^4(a)\},\{f^5(a)\}\}$$

There are two positive equality literals in P, so we merge $f^3(a)$ and a, as well as $f^5(a)$ and a:

$$\cong_1 \colon \{\{a, f^3(a), f^5(a)\},\{f(a)\},\{f^2(a)\},\{f^4(a)\}\}$$

We now look for congruences in need of merging. Looking at a and $f^5(a)$, there are no terms of $f^6(a)$ in any classes, so no congruences need to be accounted for. But a and $f^3(a)$ are also related under $\cong_1$, and because $f(a)$ and $f^4(a)$ are in different classes, we merge them.

$$\cong_2 \colon \{\{a, f^3(a), f^5(a)\},\{f(a), f^4(a)\},\{f^2(a)\}\}$$

The most recent merge implies that $f^2(a)$ and $f^5(a)$ should also be merged:

$$\cong_3 \colon \{\{a, f^2(a), f^3(a), f^5(a)\},\{f(a), f^4(a)\}\}$$

And now, because $f^2(a) \cong_3 f^3(a)$, we must merge the two remaining classes:

$$\cong_4\text{: } \{\{a, f(a), f^2(a), f^3(a), f^4(a), f^5(a)\}\}$$

This latest $\cong_4$ *must* be the congruence closure, because there are no further opportunities to merge distinct classes. Moving on to the final step, there is one negative literal $f(a) \neq a$ in P, and $f(a) \cong_4 a$, so P is unsat.

4.2 Back to the theory of arrays

To wrap things up, let's go back to where we began with EUF, namely in deciding the theory of arrays. We left off with Equation 4, which was the result of removing all write terms, explicit case-splitting to avoid introducing disjunctive formulas, and finally replacing each read term with an instance of function application:

$$\begin{array}{l} x \neq y \land f_a(i) = x \land f_a(j) = y \land f_a(j) = x \land i = i \land i = j \\ x \neq y \land f_a(i) = x \land f_a(j) = y \land f_a(j) = f_a(i) \land i = i \land i \neq j \end{array}$$

We informally concluded that neither of these formulas is satisfiable, so the original array formula must not be either. Let's have a look at what the congruence closure algorithm would come up with for either formula.

- In the $i = j$ case, the congruence closure would be $\{\{f_a(i), f_a(j), x, y\}, \{i, j\}$. Translating this finding back to the theory of arrays, a satisfying assignment would be one that equated the index at $i = j$ with the value of variables x and y, but this conflicts the first negative literal $x \neq y$.
- In the $i \neq j$ case, the congruence closure would be $\{\{f_a(i), f_a(j), x, y\}\}$. This implies a different (but not disjoint) set of possible satisfying assignments than the other case, but it still relates x and y, and so it still conflicts with the first negative literal $x \neq y$.

We learned that these closures can be computed efficiently, so once an array formula has been decomposed into a series of conjunctive cases, each case is easy to solve. However, the need to perform recursive case-splitting means that the theory of arrays is expensive to decide, and is in fact exponential in the number of read-over-write terms in a given formula. In the next lecture, we'll see how to leverage DPLL to help mitigate some of this complexity.

References