Lecture notes for: Theories: Equality and uninterpreted functions

Lindsey Kuper

October 14, 2019

These are lecture notes to accompany sections 3.1-3.4 (pp. 59-72) of Kroening and Strichman's *Decision Procedures: An Algorithmic Point of View*, second edition. See https://decomposition.al/CSE290Q-2019-09/readings.html for the full collection of notes.

Agenda

- A simple decision procedure for equality logic
- Uninterpreted functions
- A decision procedure for EUF
- Proving program equivalence with EUF

A simple decision procedure for equality logic

So last time, we started talking about the lazy paradigm of SMT solving, which involves interaction between a theory solver and a SAT solver, where the theory solver can solve formulas consisting of conjunctions of literals from a given theory, and the SAT solver does CDCL.

(This, by the way, is in contrast to eager SMT solving, which is what the STP solver does, and we'll we talking about that on Friday.)

But today, we're talking about lazy solving. Regardless of whether we're doing offline lazy solving or online lazy solving, we're going to need a theory solver. And most of Kroening and Strichman is devoted to the implementation of these theory solvers for various theories.

So today, we're talking about solvers for the theory of equality, which we've brought up already. This theory lets you write formulas where the atoms are equalities, like this:

$$y = z \wedge \neg(x = z)$$

Q: In the previous chapter, they sketched out a simple decision procedure for formulas like this. Does anyone remember what it was?

A: So they said that you can turn it into a graph problem, where every variable is a node in a graph, and then you have two kinds of edges, equality edges and inequality edges. So we'd have a graph with three nodes — a node each for x, y, and z — and an equality edge from y to z and an inequality edge from x to z. Then the idea is that the formula is unsatisfiable if and only if you have two nodes that are connected by an inequality edge and also have a *path* between them via equality edges.

So, $y = z \wedge \neg(x = z)$ would be satisfiable, right? But if you had, say,

$$y = z \wedge \neg(x = z) \wedge x = y$$

then that would be unsatisfiable, because there's an inequality edge from x to z and also an equality path from x to z.

Q: So, the book showed this just to make it clear right off the bat that a decision procedure actually exists for the conjunctive fragment of equality logic. But does anyone have any qualms about this particular decision procedure that they show?

A: My qualm about it at first was that, well, sure, this decision procedure can tell you if the formula is satisfiable or not, but if it is satisfiable, it won't tell you what the satisfying assignment is. So the constructivist mathematician in me didn't like it much. But then I figured that after you determine satisfiability, you can construct a satisfying assignment. You'd do it like this: pick a node, pick an arbitrary value from the domain to assign to it. For anything reachable via equality edges, assign them the same value. Then pick a different arbitrary value from the domain, pick another node, and keep going. It should be fine and you shouldn't violate the inequality constraints as long as you keep picking a different domain value.

It turns out it's actually a little more complicated than this, because constants from whatever domain you've chosen are allowed in equality logic, too:

$$y = z \lor (\neg(x = z) \land x = 2)$$

In the book they say you can just replace every constant with a variable, and then for any constants that were different from each other, you add a new conjunct to the formula saying that those variables have to be different. When you do this, you get a formula that is satisfiable if the original formula was satisfiable, and unsatisfiable if the original formula was unsatisfiable. (The jargon for this is *equisatisfiable*. The old and new formulas are equisatisfiable if they're both satisfiable or both unsatisfiable.) So, the claim is that we don't have to worry about solving formulas that have constants in them, because we can construct an equisatisfiable formula without the constants.

But, again, if you want to know more than just "yes, it's satisfiable" if it's satisfiable — if you actually want to know the satisfying assignment — then I think you would have to do something to keep the information around about which constants those were that you replaced with variables. It depends if "yes, it's satisfiable" is a good enough answer for you, or if you need to actually know the satisfying assignment.

Uninterpreted functions

This chapter is in fact not just about the theory of equality; it throws something else into the mix, which is the notion of *uninterpreted functions*. So now it's the "theory of equality and uninterpreted functions", or *EUF* for short.

This means that instead of just having atoms that look like $y = z$, the terms on either side of those equalities can now be function symbols that take terms as arguments. So for instance, $F(y) = z$ could be an atom.

What does F mean? We don't know what it means, and we don't care! All we know about F is that if (say) $y = y'$, then $F(y) = F(y')$. This property is called *functional congruence*: instances of the same function return the same value if given equal arguments. Having uninterpreted functions in your theory means that you only care about functional congruence; you don't care about the semantics of functions otherwise.

When you *do* care about the semantics of functions, you have to add axioms to your theory to define their semantics. And that's called having interpreted functions. But if you don't care about the semantics then you just need functional congruence.

Uninterpreted functions can also take multiple terms as arguments, and we can generalize the definiton of congruence in the expected way. So, $F(y, x) = F(z, q)$ if $y = z$ and $x = q$.

Q: What's the purpose of having uninterpreted functions in a theory?

A: It turns out that if you take all the interpreted functions in a theory and replace them with uninterpreted functions, and the formula is valid with just the uninterpreted functions, then that means it was also valid in the original version with interpreted functions. And it can be a lot easier to check for validity when you just have uninterpreted functions. So if you want to check for validity, one reasonable strategy is to convert interpreted functions to uninterpreted functions and then check for validity with an EUF solver.

Q: What's the catch?

A: Let F be a formula with interpreted functions and let F' be its EUF counterpart with uninterpreted functions (swapping any axioms that defined the semantics of functions for the functional congruence axiom), and suppose we have an EUF solver. Then:

- If the EUF solver says F' is valid, then F is valid.
- If the EUF solver says the formula is not valid, then we don't know: F might be valid, or it might not.

Q: What's an example of a time when the EUF solver might say that a formula is not valid when in fact it would have been valid if the functions were interpreted?

A: So suppose that your formula is this:

$$x_1 = y_2 \land y_1 = x_2 \land (x_1 + y_1 = x_2 + y_2)$$

So, is this formula valid? If we use the intended interpretation for $+$, then yes. But maybe we don't have a decision procedure for formulas with $+$ in them; maybe we only have an EUF decision procedure. So we translate the formula to EUF like so:

$$x_1 = y_2 \land y_1 = x_2 \land F(x_1, y_1) = F(x_2, y_2)$$

And we have functional congruence of F as part of EUF. So, is this formula valid in the EUF theory? No, it's not! Why not?

What axiom about $+$ did we lose when we switched to F? We lost the fact that $+$ is *commutative*. Functional congruence alone won't help us, because x_1 and x_2 aren't necessarily equal and y_1 and y_2 aren't necessarily equal.

Q: Did we lose soundness by switching from $+$ to F?

A: Nope! We're not going to lie and claim a formula is valid when it isn't as a result of switching from $+$ to F. But we do lose completeness. If a formula is not valid after translation to EUF, that doesn't mean it wasn't valid originally. In other words, this solving approach is sound, but it's not complete.

But, as we've discussed, sound but not complete isn't so bad. It just means that we're being conservative. And moreover, we can try checking for validity of F' and then fall back to checking validity of F if you have to. In fact, what we can do is start by trying to check for validity with the more abstract version of a formula and then *gradually* add axioms or constraints if you aren't able to establish validity that way. This is called *abstraction-refinement*.

That said, it's pretty common that you'd want to have properties like commutativity of addition expressible right from the get-go, in which case you would want to have a theory that lets you express addition. You might have heard of the theory "EUFA", which stands for "equality, uninterpreted functions, and arithmetic". We'll be reading some papers later in the course that use that theory.

A decision procedure for EUF

Back to EUF for now, though. So, how do we solve EUF formulas? The algorithm for this is called *congruence closure* and is due to Robert Shostak, who published it in 1978. (Incidentally, Robert Shostak is also responsible for much of what we know about Byzantine fault tolerance. He's also the brother of Seth Shostak from the SETI Institute who does a lot of public speaking about aliens.)

So here's the algorithm. You have a formula F that's a conjunction of equalities over variables and uninterpreted functions.

The steps of the algorithm are:

1. Put terms in an equivalence class together if you have a predicate saying they're equal; put all other terms in singleton equivalence classes
2. Merge equivalence classes that have a shared term; continue until none are left to be merged
3. (the "congruence closure" step) If terms t_i and t_j are in the same class, merge classes containing $F(t_i)$ and $F(t_j)$ for some F; continue until none are left to be merged
4. If there exists a disequality predicate $t_i \neq t_j$ such that t_i and t_j are in the same class, return "unsatisfiable", otherwise, return "satisfiable"

So, for instance, say this is your F (and this is from Kroening and Strichman, chapter 4):

$$x_1 = x_2 \land x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1 \land F(x_1) \neq F(x_3)$$

The first step is to put terms in an equivalence class together if you have a predicate saying they're equal. So, our equivalence classes are:

$$\{\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}\}$$

Now we can do step 2:

$$\{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}\}$$

And now step 3:

$$\{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\}\}$$

But since $F(x_1)$ and $F(x_3)$ are in the same class, and we had the predicate $F(x_1) \neq F(x_3)$, this formula is unsatisfiable.

It turns out that this is efficient to implement because all these subsets are disjoint, and there are data structures that are specifically designed for when you have a bunch of disjoint subsets of a set that periodically have to be merged. Those are known as *union-find* data structures because they support an operation called "union" and an operation called "find", and you can read about that on your own if you want to.

Proving program equivalence with EUF

There's a lot you can express with just EUF. There are a couple examples of this in Kroening and Strichman that under the broad heading of *program equivalence* problems. I use the term "program" broadly; by program equivalence problems we might mean:

- proving that two implementations of a data structure providing the same API — maybe a reference implementation and a fast, highly optimized implementation — are interchangeable in terms of what they compute.
- proving equivalence of two circuits, again, maybe a slow one and a fast one.
- proving that the output of a compiler (or a phase of a compiler) computes the same thing as the input to the compiler (or phase) does.

So let's just take the compiler correctness example from the book. Say we have the program:

$z = (x_1 + y_1) * (x_2 + y_2)$

And there's a phase of compilation that flattens the code to:

$u_1 = x_1 + y_1$
$u_2 = x_2 + y_2$
$z = u_1 * u_2$

which might have a more direct counterpart in assembly instructions.

So what's the formula that we want to check validity of?

We want to show that the value assigned to z you get when you run the target program implies the assignment to z you would get when your run the source program.

$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 * u_2 \implies z = (x_1 + y_1) * (x_2 + y_2)$

This formula is called a *verification condition*. Coming up with verification conditions whose truth will imply the property we want to prove is a big part of the work of automated software verification.

Q: By the way, why is this an implication and not an equivalence? And why does the implication go this way and not the other way?

A: This is a little tricky. So the thing we're trying to prove is that, whatever value of z the compiler produces, it's also a legitimate value of z according to the source code. You can think of the source code as being a specification and the target code as being one particular implementation of that specification. In general (although maybe not for this toy example), there might be multiple ways to satisfy the *specification* of the source code, and the compiler might implement only one of those ways.

So, for example, and I’m completely making this up, the semantics of the language might tolerate a little bit of floating point error in the computation of z. And whatever the compiler produces has to be within those bounds. But if the compiler wants to have *tighter* bounds on what it produces than the language spec does, that’s okay. So we want a Venn diagram that looks like this:

```
--------------------------------------------
|                                          |
|  values of z that satisfy source formula |
|                                          |
|     --------------------------------     |
|     |                              |     |
|     |   values of z that satisfy   |     |
|     |        target formula        |     |
|     |                              |     |
|     --------------------------------     |
|                                          |
--------------------------------------------
```

And in general, when you have $A \subset B$, that means $x \in A \implies x \in B$:

```
--------------------------------------------
|                                          |
|                    B                     |
|                                          |
|     --------------------------------     |
|     |                              |     |
|      |                A                 |      |
|      |                                  |      |
|      |                                  |      |
|      ------------------------------------      |
|                                                |
--------------------------------------------------
```

OK, so we want to prove that our verification condition is valid, but we only have an EUF decision procedure. We can turn those into uninterpreted functions, though.

So instead of

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 * u_2 \implies z = (x_1 + y_1) * (x_2 + y_2)$$

we have:

$$u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2) \implies z = G(F(x_1, y_1), F(x_2, y_2))$$

which is a formula that we can convert to CNF and hand off to our EUF decision procedure. If it's valid according to that decision procedure, then the original formula was valid, too.