15-414: Bug Catching: Automated Program Verification

Lecture Notes on Real-world SMT

Matt Fredrikson Ruben Martins

Carnegie Mellon University
Lecture 15

1 Introduction

In the previous lecture we studied the Nelson-Oppera procedure and the DPLL(T) framework that allows us to build decision procedures for formulas that use multiple theories. In this lecture, we will take a closer look to the theory of equality with uninterpreted functions and show how it can be used to abstract complex functions that may be hard to verify. As a real-world example, we will show how the theory of equality with uninterpreted functions can be used to prove equivalence of programs and how to use SMT solvers. Finally, we will present a decision procedure for the theory of equality with uninterpreted functions based on congruence closure. [1]

2 Preliminaries

We start by reviewing the signature and axioms of the theory of equality with uninterpreted functions.

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

consists of

- = (equality), a binary predicate;
- and all constant, function and predicate symbols.

The axioms of T_{E} are the following:

[1] Lecture notes based on [BM07] and [KS16].

1. $\forall x.x = x$ (reflexivity)
2. $\forall x, y.x = y \rightarrow y = x$ (symmetry)
3. $\forall x, y, z.x = y \wedge y = z \rightarrow x = z$ (transitivity)
4. $\forall \bar{x}, \bar{y}.(\bigwedge_{i=1}^{n} x_i = y_i) \rightarrow f(\bar{x}) = f(\bar{y})$ (congruence)
5. $\forall \bar{x}, \bar{y}.(\bigwedge_{i=1}^{n} x_i = y_i) \rightarrow (p(\bar{x}) \leftrightarrow p(\bar{y}))$ (equivalence)

Consider the Σ-formula φ

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

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 $f(a) \neq a$. More 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)

Note that even though we have the equivalence axiom, we can transform an instance of this axiom to an instance of the congruence axiom. This transformation allows us to disregard the equivalence axiom. For example, given Σ-formula:

$$x = y \rightarrow (p(x) \leftrightarrow p(y))$$

introduce a fresh constant c and a fresh function f_p, and write

$$x = y \rightarrow ((f_p(x) = c)) \leftrightarrow (f_p(y) = c))$$

In the rest of this lecture, we will consider Σ-formulae without predicates other than $=$.

3 Proving equivalence of programs

Replacing functions with uninterpreted functions in a given formula is a common technique for making it easier to reason about (e.g., to prove its validity) At the same time, this process makes the formula *weaker* which means that it can make a valid formula invalid. This observation is summarized in the following relation, where φ^{UF} is derived from a formula φ by replacing some or all of its functions with uninterpreted functions:

```c
int power3(int in)
{
  int i, out_a;
  out_a = in;
  for (i = 0; i < 2; i++)
    out_a = out_a * in;
  return out_a;
}
```

(a)

```c
int power3_new(int in)
{
  int out_b;

  out_b = (in * in) * in;

  return out_b;
}
```

(b)

Figure 1: Two C functions. We can simplify the proof of their equivalence by replacing the multiplication operator by an uninterpreted function.

$$\models \varphi^{UF} \rightarrow \varphi$$

Uninterpreted functions are widely used in calculus and other branches of mathematics, but in the context of reasoning and verification, they are mainly used for simplifying proofs. Under certain conditions, uninterpreted functions let us reason about systems while ignoring the semantics of all functions, assuming they are not necessary for the proof.

Assume that we have a method for checking the validity of a Σ-formula in T_{E}. [2] Relying on this assumption, the basic scheme for using uninterpreted functions is the following:

1. Let φ denote a formula of interest that has interpreted functions. Assume that a validity check of φ is too hard (computationally), or even impossible.

2. Assign an uninterpreted function to each interpreted function in φ. Substitute each function in φ with the uninterpreted function to which it is mapped. Denote the new formula by φ^{UF}.

3. Check the validity of φ^{UF}. If it is valid then φ is valid. Otherwise, we do not know anything about the validity of φ.

As a motivating example consider the problem of proving the equivalence of two C functions shown in Figure 1. In general, proving the equivalence of two programs is undecidable, which means there is no sound and complete to prove such an equivalence. However, in this case equivalence can be decided since the program does not have unbounded memory usage. A key observation about these programs is that they have only bounded loops, and therefore it is possible to compute their input/output relations. The derivation of these relations from these two programs can be as follows:

1. Remove the variable declarations and "return statements".

[2] Later in this lecture we will show how to check the validity of a Σ-formula in T_{E} with congruence closure.

$$
\begin{aligned}
out0\_a &= in0\_a \wedge \\
out1\_a &= out0\_a * in0\_a \wedge \\
out2\_a &= out1\_a * in0\_a
\end{aligned}
\qquad\qquad
out0\_b = (in0\_b * in0\_b) * in0\_b
$$

(a) (φ_a) (b) (φ_a)

Figure 2: Two formulas corresponding to the programs (a) and (b) in Figure 1.

2. Unroll the **for** loop.
3. Replace the left-hand side variable in each assignment with a new auxiliary variable.
4. Whenever a variable is read, replace it with the auxiliary variable that replaced it in the last place where it was assigned.
5. Conjoin all program statements.

These operations result in the two formulas φ_a and φ_b which are shown in Figure 2. This procedure to transform code into a first-order formula is known as **static single assignment** (SSA). A generalization of this form to programs with "if" branches and other constructs will be further explored in future lectures when discussing bounded model checking. For this lecture, we apply a limited form of SSA to illustrate how uninterpreted functions can be used to abstract the multiplication operator.

To show that these programs are equivalent with respect to their input-outputs, we must show that the following formula Φ is valid:

$$in0\_a = in0\_b \wedge \varphi_a \wedge \varphi_b \rightarrow out2\_a = out0\_b$$

Showing the validity of Φ is equivalent to show the unsatisfiability of $\neg\Phi$. We can show that $\neg\Phi$ is unsatisfiable by using SMT solvers.

4 Using SMT solvers

SMT solvers take as input a formula in a standardize format (SMT2-Lib format). A detailed description of the SMT2-Lib format is available at:

http://smtlib.cs.uiowa.edu

SMT solvers support a variety of theories, namely: the theory of arrays with extensionality, the theory of bit vectors with arbitrary size, the core theory defining the basic Boolean operators, the theory of floating point numbers, the theory of integer number and the theory of reals. [3]

[3] Further details on each theory are available at http://smtlib.cs.uiowa.edu/theories.shtml.

```
(declare-fun out0_a () (Int))
(declare-fun out1_a () (Int))
(declare-fun in0_a () (Int))
(declare-fun out2_a () (Int))
(declare-fun out0_b () (Int))
(declare-fun in0_b () (Int))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
       (and (= out1_a (* out0_a in0_a)) ; out1_a = out0_a * in0_a
            (= out2_a (* out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (* (* in0_b in0_b) in0_b))) ; out0_b = in0_b * in0_b *
      in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 3: SMT encoding of Φ using mathematical integers to model integers.

If you want to try SMT solving, we recommend doing the z3 tutorial at:

https://rise4fun.com/z3/tutorial

and trying z3 online at:

https://rise4fun.com/z3/

Before using SMT solvers to show that $\neg\Phi$ is unsatisfiable, we must decide how we will model integers since this will restrict the underlying theories used by the SMT solver.

4.1 Modeling integers as mathematical integers

If we model integers as mathematical integer than the SMT solver will use the theory of integers and will be able to show that both programs are equivalent. Figure 3 shows the SMT encoding of Φ when using integers: You can try this encoding online at:

https://rise4fun.com/Z3/BLQpl

However, integers are not represented as mathematical integers in C. If we want to model integers as the ones being used in C then we should model them using bit vectors (of size 32 or 64).

```
(declare-fun out0_a () (_ BitVec 512))
(declare-fun out1_a () (_ BitVec 512))
(declare-fun in0_a () (_ BitVec 512))
(declare-fun out2_a () (_ BitVec 512))
(declare-fun out0_b () (_ BitVec 512))
(declare-fun in0_b () (_ BitVec 512))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
       (and (= out1_a (bvmul out0_a in0_a)) ; out1_a = out0_a * in0_a
         (= out2_a (bvmul out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (bvmul (bvmul in0_b in0_b) in0_b))) ; out0_b = in0_b *
     in0_b * in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 4: SMT encoding of Φ using bit vectors to model integers.

4.2 Modeling integers as bit vectors

Modeling integers as bit vectors as the advantage of capturing the C model and being able to detect potential overflows. However, using the bit vector theory is not as efficient as using the theory of integers. In particular, assume we want to show that the programs are equivalent for a bit width of 512. The SMT encoding when using bit vectors is shown in the Figure 5. You can try this encoding online at:

https://rise4fun.com/Z3/ibsw3

This formula is much more challenging to be solved than the previous one and will become harder as the bit with increases. For example, if you try it online you will get an out of memory error. You can also try it in your own computer (since you should have z3 installed) by running the following command:

```
$ z3 -smt2 formula
```

where formula is a file with the contents of Figure 5. The reason for the memory blowup is the multiplication operator when using bit vectors. Can we avoid this issue altogether? What if we consider the multiplication operator as an uninterpreted function?

4.3 Using uninterpreted functions

If we consider an uninterpreted function f that takes as input two bit vectors and returns a bit vector than we can replace the bit vector multiplication operator (bvmul) by f. If we are able to prove that this formula is unsatisfiable, then we can conclude that

```
(declare-fun out0_a () (_ BitVec 512))
(declare-fun out1_a () (_ BitVec 512))
(declare-fun in0_a () (_ BitVec 512))
(declare-fun out2_a () (_ BitVec 512))
(declare-fun out0_b () (_ BitVec 512))
(declare-fun in0_b () (_ BitVec 512))
(declare-fun f ((_ BitVec 512) (_ BitVec 512)) (_ BitVec 512))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
       (and (= out1_a (f out0_a in0_a)) ; out1_a = out0_a * in0_a
            (= out2_a (f out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (f (f in0_b in0_b) in0_b))) ; out0_b = in0_b * in0_b *
      in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 5: SMT encoding of Φ using an uninterpreted function for multiplication.

the original formula is also unsatisfiable and we are able to show the equivalence between the two programs when representing integers by bit vectors of width 512. This formula is much easier to be solved than the one using bit vector multiplication since we abstracted the multiplication function and the SMT solver will not need to reason about what f does but only that it is a function. You can try this encoding online at:

https://rise4fun.com/Z3/V7Sf

But how can we show the satisfiability of a formula with equality and uninterpreted functions? In the next section, we will show a procedure based on congruence closure to determine the satisfiability of a formula with equality and uninterpreted functions.

5 Congruence closure

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.

Definition 1 (Equivalence relation)**.** A binary relation R over a set S is an equivalence relation if:

1. Reflexive: $\forall_s \in S.sRs$;
2. Symmetric: $\forall_{s_1,s_2} \in S.s_1Rs_2 \rightarrow s_2Rs_1$;
3. Transitive: $\forall_{s_1,s_2,s_3} \in S.s_1Rs_2 \wedge s_2Rs_3 \rightarrow s_1Rs_3$.

For example, the relation $=$ is an equivalence relation over real numbers and $\equiv_2$ is an equivalence relation over $\mathbb{Z}$.

Definition 2 (Congruence relation)**.** Consider a set S equipped with functions $F = \{f_1, \ldots, f_n\}$. A relation R over S is a congruence relation if it is an equivalence relation and for every n-ary function $f \in F$:

$$\forall_{\bar{s},\bar{t}} \bigwedge_{i=1}^{n} s_iRt_i \rightarrow f(\bar{s})Rf(\bar{t})$$

Definition 3 (Equivalence and congruence classes)**.** For a given equivalence relation over S, every member of S belongs to an equivalence class. The equivalence class of $s \in S$ under R is the set:

$$[s]_R \stackrel{\text{def}}{=} \{s' \in S : sRs'\}$$

If R is a congruence relation then this set is called a congruence class.

For example, the equivalence class of 1 under $\equiv_2$ are the odd numbers and the equivalence class f 6 under $\equiv_3$ the multiples of 3.

Definition 4 (Equivalence closure)**.** The equivalence closure R^E of the binary relation R over S is the equivalence relation such that:

- $R \subseteq R^E$;
- for all other equivalence relations R' s.t. $R \subseteq R'$, $R^E \subseteq R'$.

Thus, R^E is the smallest equivalence relation that includes R.

Let $S = \{a, b, c, d\}$ and $R = \{aRb, bRc, dRd\}$ then

- aRb, bRc, $dRd \in R^E$ since $R \subseteq R^E$;
- aRa, bRb, $cRc \in R^E$ by reflexivity;
- bRa, $cRb \in R^E$ by symmetry;
- $aRc \in R^E$ by transitivity;
- $cRa \in R^E$ by symmetry;

Hence,

$$R^E = \{aRb, bRa, aRz, bRb, bRc, cRb, cRc, aRc, cRa, dRd\}.$$

Definition 5 (Subterm set)**.** The subterm set S_φ of Σ-formula φ is the set that contains precisely the subterms of φ.

For example, the subterm set of φ:

$$\varphi : f(a,b) = a \wedge f(f(a,b),b) \neq a$$

is

$$S_\varphi = \{a, b, f(a,b), f(f(a,b),b)\}.$$

Now we relate the congruence closure of a Σ-formula's subterm set with its T_{E}-satisfiability. Given Σ-formula φ

$$\varphi : s_1 = t_1 \wedge \ldots \wedge s_m = t_m \wedge s_{m+1} \neq t_{m+1} \wedge \ldots \wedge s_n \neq t_n$$

with subterm set S_φ, φ is T_{E}-satisfiable iff there exists a congruence relation $\sim$ over S_φ such that

- for each $i \in \{1, \ldots, m\}, s_i \sim t_i$;
- for each $i \in \{m+1, \ldots, n\}, s_i \not\sim t_i$.

The goal of the congruence closure algorithm is to construct the congruence relation of a formula's subterm set, or to prove that no congruence relation exists. The algorithm performs the following steps:

1. Construct the congruence closure $\sim$ of

   $$\{s_1 = t_1, \ldots, s_m = t_m\}$$

   over the subterm set S_φ. Then

   $$\sim \models s_1 = t_1 \wedge \ldots \wedge s_m = t_m$$

2. If $s_i \sim t_i$ for any $i \in \{m+1, \ldots, n\}$ then φ is unsatisfiable;
3. Otherwise, $\sim \models \varphi$ and φ is satisfiable.

How do we actually construct the congruence closure in Step 1? Initially, begin with the finest congruence relation $\sim_0$ given by the partition

$$\{\{s\} : s \in S_\varphi\}$$

in which each term of S_φ is its own congruence class. Then, for each $i \in \{1, \ldots, m\}$, impose $s_i = t_i$ by merging the congruence classes

$$[s_i] \sim_{i-1} \text{ and } [t_i] \sim_{i-1}$$

to form a new congruence relation $\sim_i$. To accomplish this merging, first form the union of $[s_i] \sim_{i-1}$ and $[t_i] \sim_{i-1}$. Then propagate any new congruence that arise within this union.

Consider the Σ-formula φ

$$\varphi : f(a,b) = a \wedge f(f(a,b),b) \neq a$$

Construct the following initial partition by letting each member of the subterm set S_φ be its own class:

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

According to the first literal $f(a,b) = a$, merge

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

to form partition

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

According to the congruence axiom,

$$f(a,b) \sim a, b \sim b \text{ implies } f(f(a,b),b) \sim f(a,b),$$

resulting in the new partition

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

This partition represents the congruence closure of S_φ. Now, it is the case that

$$\{\{a,f(a,b),f(f(a,b),b)\},\{b\}\} \models \varphi \ ?$$

No! Since $f(f(a,b),b) \sim a$ but φ asserts that $f(f(a,b),b) \neq a$. Therefore, φ is T_{E}-unsatisfiable.

6 Summary

- Uninterpreted functions can be used to simplify proofs by replacing (complex) interpreted functions by uninterpreted functions;
- Let φ^{UF} be φ with all its interpreted functions replaced by uninterpreted functions. Then:
$$\models \varphi^{UF} \rightarrow \varphi$$
- We can use SMT solvers to check if two programs are equivalent:
  - If we represent integers as mathematical integers than the problem is relatively easy to be solved;
  - If we represent integers as bit vectors than the problem becomes more challenging because of bit vector multiplication;
  - If we abstract bit vector multiplication with uninterpreted functions than we can achieve scalability and prove that two programs are equivalent for any bit width.
- Congruence closure can be use to check the satisfiability of a formula with equality and uninterpreted functions.

References

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.