Program verification

Nelma Moreira

Program verification
Lecture 23
Theories: integers, arrays and bit vectors. Quantifier elimination.
Modeling and Examples in Z3

Linear arithmetic

Decision procedures for conjunctions of linear constraints.

- Domains: integers, rationals, reals
- For integers the problem is NP-complete
- Classic methods of optimization (that can be reduced to decision problems): Simplex Algorithm accepts constraints of the form

$$a_1 x_1 + \cdots + a_n x_n = 0$$

$$\ell_i \leq x_i \leq u_i$$

- Branch and Bound
- Fourier-Motzin Variable Elimination
- Omega Test: conjunction of linear constraints of the form $\sum_{i=1}^{n} a_i x_i = b$ and $\sum_{i=1}^{n} a_i x_i \leq b$ ($a_i \in \mathbb{Z}$

Theory of arrays

The axiomatization of *arrays* is the following, where the quantifier-free fragment is decidable:

$$\begin{aligned} &\mathcal{T}_E \\ &\forall a, i, j. i = j \rightarrow read(a, i) = read(a, j) \\ &\forall a, i, j, v. i = j \rightarrow read(write(a, i, v), j) = v \\ &\forall a, i, j, v. \neg(i = j) \rightarrow read(write(a, i, v), j) = read(a, j) \\ &\forall a, b. (\forall i. read(a, i) = read(b, i)) \rightarrow a = b \end{aligned}$$

We formalise an *array* a as a map from index type theory T_I to an element type theory T_E.

The type of an array a is

$$T_A = T_I \to T_E$$

Reading and writing

Let $a \in T_A$ be an array.The basic operations are:

Reading $a[i]$ denotes $read(a, i)$, i.e., an element of T_E that correspond to the index $i \in T_I$ of a

Writing $a[i \leftarrow e]$ denotes $write(a, i, e)$, i.e., $e \in T_E$ is the value to be written at index i of a.

Array Logic

We assume that T_I is a theory where the quantified fragment is decidable (e.g. Presburger arithmetic, i.e., linear arithmetic over integers).

In this way it is possible to model properties such as *there exists an array element that is zero* or *all elements of the array are nonzero*.

Let t_I and t_E be the terms of T_I and T_E and $id_a \in Var_{array}$ identifiers for arrays, then the terms for T_A are:

$$t_A := id_a \mid t_A[t_I \leftarrow t_E]$$

Terms of t_E are extended to include the elements of arrays:

$$t_E := t_A[term_I] \mid \cdots$$

The formulae include the ones of T_I and T_E plus the equality of terms of T_A., i.e.

$$\varphi := t_A = t_A \mid \cdots$$

We can consider $a_1 = a_2$ an abbreviation of $\forall i.a_1[i] = a_2[i]$, if T_I includes quantifiers. The axioms given above can be rewritten as:

$$\forall a_1 \in T_A.\forall a_2 \in T_A.\forall i \in T_I.\forall j \in T_I.((a_1 = a_2 \wedge i = j) \implies a_1[i] = a_2[j]), \quad (1)$$

$$\forall a \in T_A.\forall e \in T_E.\forall i \in T_I.\forall j \in T_I.a[i \leftarrow e][j] = \begin{cases} e & i = j, \\ a[j] & \text{otherwise}, \end{cases} \quad (2)$$

$$\forall a_1 \in T_A.\forall a_2 \in T_A.(\forall i \in T_I.a_1[i] = a_2[i]) \implies a_1 = a_2. \quad (3)$$

The axiom (2) is called **read-over write axiom** and the axiom (3) is the **extensionality rule**

Note: in this theory the arrays have unbounded dimension. The array dimension can be given using formulae over integers.

Example

Consider the Hoare triple

$$\{True\}\ \texttt{for}\ i \leftarrow 0\ \texttt{to}\ 99\ \texttt{do}\ a[i] \leftarrow 0\ \{\forall 0 \leq k < 100, a[k] = 0\}$$

Let $\eta : \forall 0 \leq k < i,\ a[k] = 0$ be the invariant and the following *tableaux*:

$\{\texttt{true}\}$
$\{0 \leq 99\}$
$\texttt{for}\ i \leftarrow 0\ \texttt{to}\ 99\ \texttt{do}$
$\{$
$\{(\forall 0 \leq k < i, a[k] = 0) \wedge 0 \leq i \wedge i \leq 99\}$
$\{\forall 0 \leq k < i + 1, a[i \leftarrow 0][k] = 0\}$ $cons_{tot}$
$a[i] \leftarrow 0$
$\{\forall 0 \leq k < i + 1, a[k] = 0\}$ ass_{tot}
$\}$
$\{\eta[100/i]\}$
$\{\forall 0 \leq k < 100, a[k] = 0\}$

Arrays as Uninterpreted functions

In the previous example we need to proof the following verification condition:

$$(\forall 0 \leq k < i, a[k] = 0) \implies \forall 0 \leq k < i + 1, a[i \leftarrow 0][k] = 0$$

Suppose that there are no quantifiers over arrays, i.e. arrays are ground terms

Considering a a function we can substitute each of its instances by an uninterpreted function, where the index is the only argument.

In particular, the axiom (1) corresponds to the functional congruence.

Example 1. If T_E is the theory of strings

$$(i = j \wedge a[j] = "z") \implies a[i] =' z'$$

can be substituted by

$$(i = j \wedge F_a(j) =' z') \implies F_a(i) =' z'$$

that can be evaluated by the decision procedures already considered.

Array updates

To replace terms of the form

$$a[i \leftarrow e],$$

fresh variables of type array are introduced, $a' \in Var_{array}$ and two constraints are added (that correspond to the two cases of the read-over write rule).

This rule is an equivalence-preserving transformation

Write rule

1. $a'[i] = e$
2. $\forall j \neq i . a'[j] = a[j]$

Example 2. The formula

$$a[i \leftarrow e][i] \geq e$$

is transformed into

$$a'[i] = e \implies a'[i] \geq e$$

. The formula $a[0] = 10 \implies a[1 \leftarrow 20][0] = 10$ is transformed into:

$$(a[0] = 10 \wedge a'[1] = 20 \wedge (\forall j \neq 1 . a'[j] = a[j])) \implies a'[0] = 10.$$

Introducing F_a and $F_{a'}$ we have

$$(F_a(0) = 10 \wedge F_{a'}(1) = 20 \wedge (\forall j \neq 1 . F_{a'}(j) = F_a(j))) \implies F_{a'}(0) = 10.$$

A Reduction Algorithm for Array Logic

- The combination of Presburger theory with uninterpreted functions is in general undecidable.
- Thus, we need to restrict the set of formulas we consider.
- We consider formulae that are Boolean combinations of *array properties*.

Array properties

Definição 23.1 (Array properties)**.** *Is a formula of the form*

$$\forall i_1 \cdots \forall i_k \in T_I . \varphi_I(i_1, \ldots, i_k) \implies \varphi_V(i_1, \ldots, i_k)$$

where

1. *φ_I is called the index guard and must follow the grammar*

$$\begin{aligned} \varphi_I &:= \varphi_I \wedge \varphi_I \mid \varphi_I \vee \varphi_I \mid t_i \leq t_i \mid t_i = t_i \\ t_i &:= i_1 \mid \cdots \mid i_k \mid t \\ t &:= n \in \mathbb{N} \mid n \times id_i \mid t + t \end{aligned}$$

   Terms t are expressions over integers and id_i is a variable of T_I distinct from i_j.

2. *Index variables $i_1, \ldots, i_k$ can only be used in array read expressions of the form $a[i_j]$ in φ_V.*

Examples

- Extensionality is an array property

$$\forall i.a_1[i] = a_2[i]$$

  where the guard is true.

- The formula $a' = a[i \leftarrow 0]$ is replaced by two formulas:
  - $a'[i] = 0$ is an array property and
  - $\forall j \neq i.a'[j] = a[j]$.
    In this case we need to replace it by

$$\forall j.((j \leq i-1 \vee i+1 \leq j) \implies a'[j] = a[j])$$

    which is an array property.

A Reduction Algorithm

We now describe an algorithm that accepts a formula from the array property fragment of array theory and reduces it to an equisatisable formula that uses the element and index theories combined with equalities and uninterpreted functions.

The input will be an array property in NNF, where universal quantifiers can be replaced by existential quantifiers but no alternation of quantifiers occur (due to the syntactic restrictions).

Array-reduction

Input:	An array property formula φ_A in NNF
Output:	A formula φ^{uf} of the theories T_I and T_E, and with uninterpreted functions.

1. Apply the write rule to remove all array updates $a[i \leftarrow e]$ from φ_A .
2. Replace all existencial quantifiers $\exists i \in T_I.P(i)$ by $P(j)$, where j is a fresh variable.
3. Replace all universal quantifiers $\forall i \in T_I.P(i)$ by

$$\bigwedge_{i \in \mathcal{I}(\varphi)} P(i).$$

4. Replace the array read operators ($a[i]$) by uninterpreted functions, and obtain φ^{uf}.
5. **return** φ^{uf}.

$\mathcal{I}(\varphi)$

The set $\mathcal{I}(\varphi)$ denotes the index expressions that i might possibly be equal to in the formula φ which is the current formula. Contains:

1. All expressions used as an array index in φ expect quantified variables
2. All expressions used inside index guards in φ expect quantified variables
3. if φ contains none of the above $\mathcal{I}(\varphi) = \{0\}$ (in order to obtain a nonempty set of index expressions).

Example

Let $k, i \in \mathbb{N}_0$, and let us prove the validity of

$$(\forall k.k < i \implies a[k] = 0) \implies (\forall k.k \leq i \implies a[i \leftarrow 0][k] = 0)$$

For that we consider that its negation is not satisfiable.

$$(\forall k.k < i \implies a[k] = 0) \wedge (\exists k.k \leq i \wedge a[i \leftarrow 0][k] \neq 0)$$

By applying the write rule, we obtain

$$\begin{aligned}&(\forall k.k < i \implies a[k] = 0) \wedge a'[i] = 0 \wedge (\forall j \neq i.a'[j] = a[j])\\ &\wedge (\exists k.k \leq i \wedge a'[k] \neq 0)\end{aligned}$$

We instantiate k with k_1 to eliminate the quantifier $\exists k$

$$\begin{aligned}&(\forall k.k < i \implies a[k] = 0) \wedge a'[i] = 0 \wedge (\forall j \neq i.a'[j] = a[j])\\ &\wedge k_1 \leq i \wedge a'[k_1] \neq 0\end{aligned}$$

We have $\mathcal{I} = \{i, k_1\}$. Then we eliminate the universal quantifiers:

$$\begin{aligned}&(i < i \implies a[i] = 0) \wedge (k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0\\ &\wedge (i \neq i \implies a'[i] = a[i])\\ &\wedge (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0\end{aligned}$$

Simplifying, we get

$$\begin{aligned}&(k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0\\ &\wedge (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0\end{aligned}$$

We replace a and a' by uninterpreted functions and obtain

$$(k_1 < i \implies F_a(k_1) = 0) \wedge F_{a'}(i) = 0$$
$$\wedge (k_1 \neq i \implies F'_a(k_1) = F_a(k_1)) \wedge k_1 \leq i \wedge F'_a(k_1) \neq 0$$

Considering the three cases $k_1 < i$, $k_1 = i$ and $k_1 > i$ we can conclude that the formula is unsatisfiable.

Thus, we conclude the validity of the initial verification condition.

Arrays em SMT-LIB/Z3

- To define arrays one use the (*sort*) `Array`

```
A = Array('A', IntSort(), IntSort())
x, y = Consts('x y',IntSort())
solve(A[x] == x, Store(A, x, y) == A)
```

- $A[x]$ is defined by `Select(A,x)` (or `A[x]`)
- `Store(A,x,v)`, corresponds to $A[x \leftarrow v]$.
- `K(Sort,v)` is an array of `Sort` where all indexes have the value v (constant array, it is used to show a solution).
- For the verification condition above

```
solve (Implies(ForAll([x],(Implies(x< y, A[x]==0))),
 ForAll([x],(Implies(x<= y, Store(A, y, 0)[x]==0)))))
```

Arrays can be represented by λ-terms : if $f : A \times B \to C$ then `Lambda [x,y]. f(x, y)` has type `Array(A,B,C)`.

```
a[i]             # select array 'a' at index 'i'
                 # Select(a, i)
Store(a, i, v)   # update array 'a' with 'v' at index 'i'
                 # = Lambda(j, If(i == j, v, a[j]))
K(D, v)          # constant Array(D, R), where R is sort of 'v'.
                 # = Lambda(j, v)
Map(f, a)        # map function 'f' on values of 'a'
                 # = Lambda(j, f(a[j]))
Ext(a, b)        # Extensionality
                 # Implies(a[Ext(a, b)] == b[Ext(a, b)], a == b)
```

Bit-vector theories

Quantifier elimination

- Many of the theories considered have undecidable fragments when quantifiers are considered.
- Even if those fragments are decidable complexity of the decision procedures is high.
- For quantified boolean formulas (QBF) quantifier elimination is decidable (PSPACE-complete)
- If there are only existential quantifiers (it is a formula in prenex normal form) one can use Skolemization to obtain a equisatisfiable formula
- Some kinds of alternation of quantifiers can also yield decidable fragments of some theories.

General quantification- Skolemization and Instantiation

- Formulae in prenex normal form
- and in Skolem normal form. Application of Skolemization
- For instance, $\forall y_1 \forall y_2 \exists x.(f(y_1, y_2) \wedge f(x, y_2) \wedge x < 0)$ after Skolemization becomes:

$$\forall y_1 \forall y_2 (f(y_1, y_2) \wedge f(f_x(y_1, y_2), y_2) \wedge f_x(y_1, y_2) < 0)$$

- The general problem is to have a ground formula G which validity must be proven with respect to axioms.
- for instance, prove that

$$f(h(a), b) = f(b, h(a))$$

  is implied by

$$\forall x \forall y. f(x, y) = f(y, x)$$

- Considering the satisfiability problem we need to show that the following formula isunsatisfiable:

$$\forall x \forall y. f(x, y) = f(y, x) \wedge f(h(a), b) \neq f(b, h(a)).$$

- It is easy to see that if x is instantiated to $h(a)$ and y to b we get a contradiction.

E-matching

- Let $\forall \overline{x}.\psi \wedge G$ be the formula we want to proof unsatisfiable, with G ground.
- One can instantiate $\overline{x}$ with all ground terms of G of the same type
- But that is in general exponential.
- The solver SIMPLIFY implemented an heuristic called E-graph algorithm that is now widely used (and improved):
- for each $\forall \overline{x}.\psi$, identify those subterms in ψ that contain references to all the variables in $\overline{x}$. These are the **triggers**.
- In the example above both $f(x, y)$ and $f(y, x)$ are triggers.
- Try to match each trigger tr (pattern) to an existing ground term gr in G and take the correspond substitution. In the example, matching $f(x, y)$ to $f(h(a), b)$ yields $\overline{s} = \{x \mapsto h(a), y \mapsto b\}$.
- Assign $G := G \wedge \psi[\overline{x} \leftarrow \overline{s}]$ and check the satisfiability of G.

Example 3. Let G be

$$b = c \implies f(h(a), g(c)) = f(g(b), h(a))$$

where f is commutative, i.e.,

$$\forall x \forall y. f(x, y) = f(y, x)$$

Consider the trigger $f(x, y)$ which can match both $f(h(a), g(c))$, with substitution $\{x \mapsto h(a), y \mapsto g(c)\}$ and $f(g(b), h(a))$ with substitution $\{x \mapsto g(b), y \mapsto h(a)\}$. Then one needs to check the satisfiability of

$$b = c \wedge f(h(a), g(c)) \neq f(g(b), h(a)) \wedge$$
$$f(h(a), g(c)) = f(g(c), h(a)) \wedge f(g(b), h(a)) = f(h(a), g(b))$$

which is unsatisfiable.

E-matching algorithm

- Frequently, however, the predicates necessary for proving unsatisability are not based on terms in the existing formula.
- *Simplify* has a more flexible matching algorithm,
- which exploits its current knowledge on equivalences among various terms, which is called *E-matching*.

Algorithm 9.5.1: E-MATCHING

Input: Trigger tr, term gr, current substitution set sub
Output: Substitution set sub such that for each $\alpha \in sub$, $E \models \alpha(tr) = gr$.

1. **function** MATCH(tr, gr, sub)
2. **if** tr is a variable x **then**
3. **return**
$\{\alpha \cup \{x \mapsto gr\} \mid \alpha \in sub, x \notin dom(\alpha)\} \cup$
$\{\alpha \mid \alpha \in sub, find(\alpha(x)) = find(gr)\}$
4. **if** tr is a constant c **then**
5. **if** $c \in class(gr)$ **then return** sub
6. **else return** $\emptyset$
7. **if** tr is of the form $f(p_1, \ldots, p_n)$ **then return**

$$\bigcup_{f(gr_1,\ldots,gr_n)\in class(gr)} \begin{array}{l} \text{MATCH}(p_n, gr_n, \\ \quad \text{MATCH}(p_{n-1}, gr_{n-1}, \\ \quad\quad \ddots \\ \quad\quad\quad \text{MATCH}(p_1, gr_1, sub)\ldots)) \end{array}$$

The algorithm uses union-find to represent the classes of equivalence of equality of terms.

- $dom(\alpha)$ is the domain of the substitution
- $find(gr)$ returns the representative element of the class of gr. If two terms gr_1, gr_2 are such that $find(gr_1) = find(gr_2)$ then it means that they are equivalent.
- $class(gr)$ returns the equivalence class of gr.
- The algorithm uses functional congruence with a member of the equivalence class.
- The output of this algorithm is a set of substitutions, each of which brings us from tr to gr, possibly by using congruence closure.
- If E denotes the equalities, then for each possible substitution $\alpha \in sub$, it holds that $E \models \alpha(tr) = gr$, where $\alpha(tr)$ denotes the substitution applied to the trigger tr.
- For example, for $tr = f(x)$ and $gr = f(a)$, if $E = \{a = b\}$, the value of sub at the end of the algorithm will be $\{x \mapsto a, x \mapsto b\}$.

Example 4. Let $(\forall x.f(x) = x) \wedge (\forall y_1.\forall y_2.g(g(y_1, y_2), y_2) = y_2) \wedge g(f(g(a, b)), b) \neq b$.

The triggers are $f(x)$ and $g(y_1, y_2)$. For the first we consider

$$\text{MATCH}(f(x), f(g(a, b)), \emptyset)$$

Line 7 is invoked and we consider $g(a, b)$:

$$\text{MATCH}(x, g(a, b), \emptyset) = \{x \mapsto g(a, b)\}$$

and $f(g(a, b)) = g(a, b)$ is added to E. For the second trigger $g(g(y_1, y_2), y_2)$, the candidate ground terms for matching are $g(a, b)$ and $g(f(g(a, b)), b))$. In the first case the matching fails

$$\text{MATCH}(y_2, b, \text{MATCH}(g(y_1, y_2), a, \emptyset)) == fail$$

as $class(a)$ has no term with functional symbol g.

In the second case we have

$$\begin{aligned}
&= \text{MATCH}(y_2, b, \text{MATCH}(g(y_1, y_2), f(g(a, b)), \emptyset)) \\
&= \text{MATCH}(y_2, b, \text{MATCH}(g(y_1, y_2), g(a, b), \emptyset)) \\
&= \text{MATCH}(y_2, b, \text{MATCH}(y_2, b, \text{MATCH}(y_1, a, \emptyset))) \\
&= \text{MATCH}(y_2, b, \text{MATCH}(y_2, b, \{y_1 \mapsto a\})) \\
&= \text{MATCH}(y_2, b, \{y_1 \mapsto a, y_2 \mapsto b\}) \\
&= \{y_1 \mapsto a, y_2 \mapsto b\}
\end{aligned}$$

Note the switch between $f(g(a, b))$ and $g(a, b)$: it happens, because these two terms are in the same equivalence class according to the E-graph.

As E-matching works only with functional congruence. It cannot deal with interpreted functions (as arithmetic ones).

1 Bibliografia

References

[BdM15] Nikolai Bjorner and Leonardo de Moura. *Z3 Theorem Prover*. Rise, Microsft, 2015.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Verlag, 2007.

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