Program verification

Nelma Moreira

Departamento de Ciência de Computadores da FCUP

Lecture 11
Equality Logic and Theory of Uninterpreted Functions

- functional terms are added to the equality theory

$$\varphi^{uf} := \varphi^{uf} \wedge \varphi^{uf} \mid \neg\varphi^{uf} \mid t = t$$
$$t := x|c|F(t_1, \ldots, t_n)$$

- only functional congruence

$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \rightarrow F(x_1, \ldots, x_n) = F(y_1, \ldots, y_n)$$

- Functions of a given theory can be substituted by uninterpreted functions simplifying the validity proofs although equivalence is not preserved. We have

$$\models \varphi^{uf} \implies \models \varphi$$

- but if $\not\models \varphi^{uf}$ nothing can be concluded.

Example: Program equivalence

```
int power3(int in) {
    out = in;

    for(i=0; i<2; i++)
       out = out * in;

    return out;
}
```

```
int power3_new(int in) {
    out = (in*in)*in;
    return out;
}
```

Static single assignment (SSA) form:

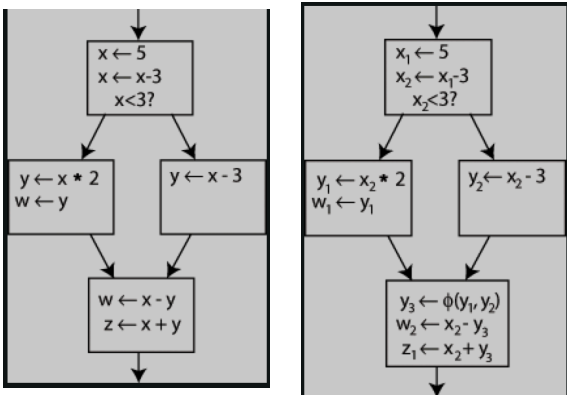
$$out_1 = in \wedge$$
$$out_2 = out_1 * in \wedge$$
$$out_3 = out_2 * in$$

$$out'_1 = (in * in) * in$$

Prove that both functions return the same value:

$$out_3 = out'_1$$

1. Remove the variable declarations and `return` statements.
2. Unroll the `for` loop.
3. Replace the left-hand side variable in each assignment with a new auxiliary variables
4. Whenever a variable is read (referred to in an expression), replace it with the auxiliary variable that replaced it in the last place where it was assigned.
5. If your program have branchs where the same variable was assigned (x_i and x_j) add an assigment $x_k := \phi(x_i, x_j)$ in the join point.
6. Conjoin all program statements.


x ← 5
x ← x-3
x<3?
y ← x * 2
w ← y
y ← x - 3
w ← x - y
z ← x + y
x1 ← 5
x2 ← x1-3
x2<3?
y1 ← x2 * 2
w1 ← y1
y2 ← x2 - 3
y3 ← ϕ(y1, y2)
w2 ← x2 - y3
z1 ← x2 + y3

In the example, given two programs we obtain two formulae φ_1 and φ_1' and we want to prove that

$$\varphi_1 \wedge \varphi_1' \implies out_3 = out_1'$$

Static single assignment (SSA) form:

$$\begin{aligned} &out_1 = in \wedge \\ &out_2 = out_1 * in \wedge \\ &out_3 = out_2 * in \end{aligned} \qquad out_1' = (in * in) * in$$

With uninterpreted functions:

$$\begin{aligned} &out_1 = in \wedge \\ &out_2 = F(out_1, in) \wedge \\ &out_3 = F(out_2, in) \end{aligned} \qquad out_1' = F(F(in, in), in)$$

The advantage is that it is easy to prove the validity of uninterpreted functions. In the example multiplication $*$ is substituted by an uninterpreted function F and we obtain φ_1^{uf} and $\varphi_1'^{uf}$.

Decision procedures for conjunctions of equalities and with uninterpreted functions with congruence closure

Input: conjunction of literals φ^{uf}
Output: Satisfiable or Unsatisfiable

1. Build congruence-closed equivalence classes.
   a) Initially, put two terms t_1, t_2 (either variables or uninterpreted function instances) in their own equivalence class if $(t_1 = t_2)$ is a predicate in φ^{uf}. All other variables form singleton equivalence classes.
   b) Given two equivalence classes with a shared term, merge them. Repeat until there are no more classes to be merged.
   c) Compute the congruence closure: given two terms t_i, t_j that are in the same class and that $F(t_i)$ and $F(t_j)$ are terms in φ^{uf} for some uninterpreted function F, merge the classes of $F(t_i)$ and $F(t_j)$. Repeat until there are no more such instances.
2. If there exists a disequality $t_i \neq t_j$ in φ^{uf} such that t_i and t_j are in the same equivalence class, return "Unsatisfiable". Otherwise return "Satisfiable"

Ex.

Let φ^{uf} be a conjunction

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

Initially the equivalence classes are:

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

We merge equal terms in the same classe

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

By the congruence closure we have:

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

Finally as $F(x_1) \neq F(x_3) \in \varphi^{uf}$, the output is Unsatisfiable.

- This algorithm can be implemented efficiently with a union-find data structure, which results in a time complexity of $O(n \log(n))$.
- To extend to general quantifier-free formulae one can use DPLL(T) or (*lazy*) variants; or eager algorithms that reduces the whole formula φ^{uf} to a equisatisfiable propositional formula (*eager* approach).

Ackerman reduction of uninterpreted functions to equality logic

We add explicit constraints to a formula in order to enforce functional consistence. Given a formula φ^{uf} we transform it in a formula from the equality logic φ^{E} such that

$$\varphi^{E} := FC^{E} \implies flat^{E}$$

where FC^{E} is a conjunction of functional-consistency constraints and $flat^{E}$ is a a version of φ^{UF} where each function instance is replaced by a new variable.

Example

Suppose we want to check

$$x_1 \neq x_2 \vee F(x_1) = F(x_2) \vee F(x_1) \neq F(x_3)$$

for validity.

1. First number the function instances:

$$x_1 \neq x_2 \vee F_1(x_1) = F_2(x_2) \vee F_1(x_1) \neq F_3(x_3)$$

2. Replace each function with a new variable:

$$x_1 \neq x_2 \vee f_1 = f_2 \vee f_1 \neq f_3$$

3. Add functional consistency constraints:

$$\left(\begin{array}{ll} (x_1 = x_2 \rightarrow f_1 = f_2) & \wedge \\ (x_1 = x_3 \rightarrow f_1 = f_3) & \wedge \\ (x_2 = x_3 \rightarrow f_2 = f_3) & \end{array} \right) \rightarrow$$

$$((x_1 \neq x_2) \vee (f_1 = f_2) \vee (f_1 \neq f_3))$$

Ackerman reduction (unary functions)

Input: An $\mathcal{EUF}$ formula φ^{uf} with m instances of an uninterpreted function F
Output: An equality logic formula φ^{E} such that φ^{E} is valid if and only if φ^{uf} is valid

1. Assign indices to the uninterpreted-function instances from subexpressions F, F_i outwards. Denote by F_i the instance of F that is given the index i, and by $arg(F_i)$ its single argument.
2. Let $flat^{E} = \mathcal{T}(\varphi^{uf})$, where $\mathcal{T}$ is a function that takes an $\mathcal{EUF}$ formula (or term) as input and transforms it to an equality formula (or term, respectively) by replacing each uninterpreted-function instance F_i with a new term-variable f_i (for nested functions, only the variable corresponding to the most external instance remains).
3. Let FC^{E} denote the following conjunction of functional-consistency constraints:

$$FC^{E} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} \mathcal{T}(arg(F_i)) = \mathcal{T}(arg(F_j)) \implies f_i = f_j$$

4. Let $\varphi^{E} := FC^{E} \implies flat^{E}$
5. Return φ^{E}

Example: equivalence of programs (cont.)

With numbered uninterpreted functions:

$$\begin{array}{l} out_1 = in \wedge \\ out_2 = F_1(out_1, in) \wedge \\ out_3 = F_2(out_2, in) \end{array} \qquad out'_1 = F_4(F_3(in, in), in)$$

Ackermann's reduction:

$$\varphi_a^E : \begin{array}{l} out_1 = in \wedge \\ out_2 = f_1 \wedge \\ out_3 = f_2 \end{array} \qquad \varphi_b^E : out'_1 = f_4$$

The verification condition:

$$\left[\left(\begin{array}{lll} (out_1 = out_2 & \rightarrow f_1 = f_2) & \wedge \\ (out_1 = in & \rightarrow f_1 = f_3) & \wedge \\ (out_1 = f_3 & \rightarrow f_1 = f_4) & \wedge \\ (out_2 = in & \rightarrow f_2 = f_3) & \wedge \\ (out_2 = f_3 & \rightarrow f_2 = f_3) & \wedge \\ (in = f_3 & \rightarrow f_3 = f_4) & \end{array} \right) \wedge \varphi_a^E \wedge \varphi_b^E \right] \longrightarrow out_3 = out'_1$$

Let φ be

$$x_1 = x_2 \implies F(F(G(x_1)))) = F(F(G(x_2)))$$

Consider the propositional variables g_1, g_2, f_1, f_2, f_3, and f_4

$$x_1 = x_2 \implies \underbrace{F(\overbrace{G(x_1)}^{g_1})}_{f_2} = \underbrace{F(\overbrace{G(x_2)}^{g_2})}_{f_4}$$

then $flat^E : x_1 = x_2 \implies f_2 = f_4$ and FC^E is

$$
\begin{aligned}
x_1 = x_2 &\implies g_1 = g_2 \\
g_1 = f_1 &\implies f_1 = f_2 \\
g_2 = f_3 &\implies f_3 = f_4 \\
g_1 = g_2 &\implies f_1 = f_3 \\
g_1 = f_3 &\implies f_1 = f_4 \\
f_1 = g_2 &\implies f_2 = f_3 \\
f_1 = f_3 &\implies f_2 = f_4 \\
g_2 = f_3 &\implies f_1 = f_4
\end{aligned}
$$

Thus, we have $\varphi^E = FC^E \implies flat^E$.

One can generalise this algorithm to functions with any arity.

Exerc.

Verifying the compilation process.Suppose the following statement:

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

which is compiled to the following sequence of three assigments:

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

The verification condition to ensure correctness is

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

Obtain a an equality formula using uninterpreted functions and applying the reduction of Ackermann. $\diamond$

We will see how to construct a formula of propositional logic that is equisatisfiable to a formula of equational logic without quantifiers.

The SAT solver is called only once.

The presented algorithm will not be very efficient but can be optimized in order to execute in polynomial time and obtain a propositional formula with a cubic size in the number of variables of the equational formula.

Assume a equational formula φ^E (without constants) with Boolean operations in NNF.

- Let $E_=$ be the set of positive literals in φ^E
- Let $E_\neq$ be the set of negative literals in φ^E

For example φ^E

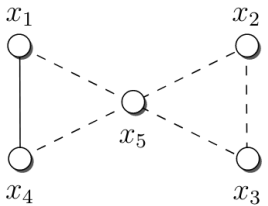
$$\begin{array}{c}(x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2\end{array}$$

We have

$$\begin{array}{rcl}E_= & = & \{f_1 = f_2, g_1 = g_2, u_1 = f_1, u_2 = f_2, z = g_1\} \\ E_\neq & = & \{x_1 \neq x_2, y_1 \neq y_2, u_1 \neq f_1, u_2 \neq f_2, z \neq g_2\}\end{array}$$

Given a equality logic formula φ^E in NNF, the equality graph of φ^E, $G^E(\varphi^E)$ is the graph $(V, E_=, E_\neq)$ where the nodes V are the variables in φ^E, the edges $E_=$ correspond to the set of positive literals and the edges $E_\neq$ to the set of negative literals.

For example, for $E_= = \{x_1 = x_5, x_2 = x_3, x_2 = x_5, x_4 = x_5\}$ e $E_\neq = \{x_1 \neq x_4\}$ we have



As in the case of conjunctions of literals, graphically we represent with a dashed line the edges that correspond to equalities and solid those of inequalities.

- The equational graph $G^E(\varphi^E)$ is an abstraction of φ^E
- It actually represents all formulas that have the same literals as φ^E
- Since it does not consider Boolean connectives, it can represent both satisfiable and unsatisfiable formulas
- For example $x_1 = x_2 \wedge x_1 \neq x_2$ and $x_1 = x_2 \vee x_1 \neq x_2$ are represented by the same graph.

- A equality path in G^E is a path with only edges of $E_=$. If there is such a path between x and y we say that $x =^* y$, for $x, y \in V$.
- A disequality path in G^E is a path with edges of $E_=$ and only one edge og $E_{\neq}$. If there is such a path between $x$and y we write $x \neq^* y$ $x, y \in V$.
- Any of these paths is simple if has no cycles.
- If $x =^* y$ it can happen that x and y have the same value but is not necessary (because we do not have the structure of the Boolean formula).
- For $x \neq^* y$ it can happen that x and y have different values
- in the example $x_1 =^* x_4$ and $x_1 \neq^* x_4$ but that may not be inconsistent
- A contradictory cycle is a cycle in G^E that has exactly one edge in $E_{\neq}$
- For $x, y \in V$ ia a contradictory cycle we have $x =^* y$ and $x \neq^* y$.
- The conjunction of literals of the cycle is unsatisfiable.
- In the example x_1, x_2, x_4 is a contradictory cycle

We can simplify formulas if there are literals that do not participate in contradictory cycles (simple).

Algorithm 11.4.1: SIMPLIFY-EQUALITY-FORMULA

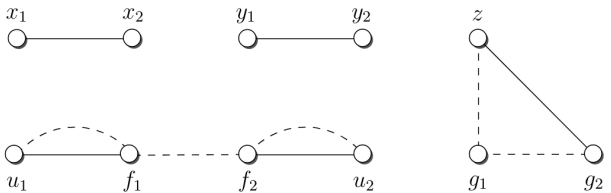
Input: An equality formula φ^{E}
Output: An equality formula $\varphi^{\mathrm{E}\prime}$ equisatisfiable with φ^{E}, with size less than or equal to the length of φ^{E}

1. Let $\varphi^{\mathrm{E}\prime} := \varphi^{\mathrm{E}}$.
2. Construct the equality graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}\prime})$.
3. Replace each pure literal in $\varphi^{\mathrm{E}\prime}$ whose corresponding edge is not part of a simple contradictory cycle with TRUE.
4. Simplify $\varphi^{\mathrm{E}\prime}$ with respect to the Boolean constants TRUE and FALSE (e.g., replace $\mathrm{TRUE} \vee \phi$ with TRUE, and $\mathrm{FALSE} \wedge \phi$ with FALSE).
5. If any rewriting has occurred in the previous two steps, go to step 2.
6. Return $\varphi^{\mathrm{E}\prime}$.

Let

$$\varphi^E := (x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2$$

the graph G^E is



The edges $f_1 = f_2$, $x_1 \neq x_2$ and $y_1 \neq y_2$ are not part of any simple contradictory cycle and can therefore be substituted by true.

$$\begin{aligned}\varphi'^E :=& (\text{true} \vee \text{true} \vee \text{true}) \wedge (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ & (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2\end{aligned}$$

Simplifying

$$\varphi'^E := (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge (u_1 = f_1 \vee u_2 = f_2 \vee z = g_1) \wedge z \neq g_2$$

And in this case, if we calculate the graph, we see that we can not simplify any further. However, if the contradictory cycles disappear, we can conclude that the formula is satisfiable (and only by simplifying).

Nonpolar equality graph

Let φ^E be a equational formula, a nonpolar equality graph of φ^E, $G^E_{NP}(\varphi^E)$ is a graph (V, E) where V are the variables of φ^E and the edges E correspond to $At(\varphi^E)$, i.e., all atomic formulae (equalities) φ^E.

- Note that $x_1 \neq x_2$ is an abbreviation of $\neg x_1 = x_2$, then G^E_{NP} only $x_1 = x_2$ is present in E.
- Instead of literals we only consider equalities (omitting the polarity).

Given φ^E the procedure generates two propositional formulas $e(\varphi^E)$ and $\mathcal{B}_{trans}$ such that

$$\varphi^E \text{ is satisfiable } \iff e(\varphi^E) \wedge \mathcal{B}_{trans} \text{ is satisfiable}$$

- The formula $e(\varphi^E)$ is the propositional skeleton of φ^E, where every predicate $x_i = x_j$ ($i \leq j$) is replaced with a new Boolean variable $e_{i,j}$
- The formula $\mathcal{B}_{trans}$ is a conjunction of implications, the transitive constraints. Each such implication is associated with a cycle in the nonpolar equality graph G^E_{NP}.
- For a cycle with n edges $\mathcal{B}_{trans}$ forbids an assignment false to one of the edges when all the other edges are assigned true.

- If φ^E is satisfiable $e(\varphi^E)$ is also satisfiable
- The constraints $\mathcal{B}_{trans}$ are enough to ensure that φ^E is satisfiable if $e(\varphi^E)$ is.

Let $\varphi^E := x_1 = x_2 \wedge (((x_2 = x_3) \wedge (x_1 \neq x_3)) \vee (x_1 \neq x_2))$ then

$$e(\varphi^E) := e_{1,2} \wedge (((e_{2,3} \wedge (\neg e_{1,3})) \vee (\neg e_{1,2}))$$

The formulae $x_1 = x_2$, $x_2 = x_3$ and $x_1 = x_3$ form a cycle in G^E_{NP} then the transitive constraints are:

$$\begin{aligned}\mathcal{B}_{trans} :=& ((e_{1,2} \wedge e_{2,3}) \implies e_{1,3}) \wedge \\ & (e_{1,2} \wedge e_{1,3}) \implies e_{2,3}) \wedge \\ & (e_{2,3} \wedge e_{1,3}) \implies e_{1,2}).\end{aligned}$$

A nonch

Algorithm 11.5.1: EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC

Input: An equality formula φ^{E}
Output: A propositional formula equisatisfiable with φ^{E}

1. Construct a Boolean formula $e(\varphi^{\mathrm{E}})$ by replacing each atom of the form $x_i = x_j$ in φ^{E} with a Boolean variable $e_{i,j}$.
2. Construct the nonpolar equality graph $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$.
3. Make $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$ chordal.
4. $\mathcal{B}_{trans} :=$ TRUE.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{i,k})$ in $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$,

$$\begin{aligned} \mathcal{B}_{trans} := \mathcal{B}_{trans} \wedge \\ (e_{i,j} \wedge e_{j,k} \implies e_{i,k}) \wedge \\ (e_{i,j} \wedge e_{i,k} \implies e_{j,k}) \wedge \\ (e_{i,k} \wedge e_{j,k} \implies e_{i,j}) \,. \end{aligned} \tag{11.42}$$

6. Return $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans}$.

- The algorithm can have exponential complexity because the number of cycles in a graph can be exponential
- A chord in a cycle is any edge that connects two nonadjacent vertices in a cycle
- Bryant et al shown that

  It is sufficient to add transitive constraints for simple chord-free cycles

- Chordal graphs A chordal graph is an undirected graph in which no cycle of size 4 or more is chord-free.
- Every graph can be made chordal in a time polynomial in the number of vertices
- Since the only chord-free cycles in a chordal graph are triangles, this implies that applying the procedure to these graphs can be done in polynomial time and obtain a formula whose size is not more than cubic in the number of variables (3 constraints for each triangle). The newly added chords are represented by new variables that appear in $\mathcal{B}_{trans}$ but not in $e(\varphi^E)$.

Algorithm 11.5.1: EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC

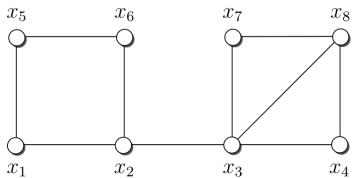
Input: An equality formula φ^{E}
Output: A propositional formula equisatisfiable with φ^{E}

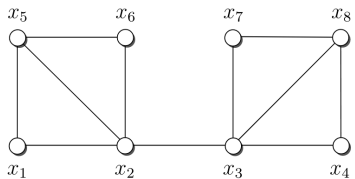
1. Construct a Boolean formula $e(\varphi^{\mathrm{E}})$ by replacing each atom of the form $x_i = x_j$ in φ^{E} with a Boolean variable $e_{i,j}$.
2. Construct the nonpolar equality graph $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$.
3. Make $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$ chordal.
4. $\mathcal{B}_{trans} :=$ TRUE.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{i,k})$ in $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$,

$$\begin{aligned} \mathcal{B}_{trans} := \mathcal{B}_{trans} \wedge \\ (e_{i,j} \wedge e_{j,k} \implies e_{i,k}) \wedge \\ (e_{i,j} \wedge e_{i,k} \implies e_{j,k}) \wedge \\ (e_{i,k} \wedge e_{j,k} \implies e_{i,j}) \,. \end{aligned} \tag{11.42}$$

6. Return $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans}$.

A nonchordal nonpolar equality graph corresponding to φ^E and a possible chordal version of it (right).





For the triangle (x_1, x_2, x_5),

$$
\begin{aligned}
e_{1,2} \wedge e_{2,5} &\implies e_{1,5} \ , \\
e_{1,5} \wedge e_{2,5} &\implies e_{1,2} \ , \\
e_{1,2} \wedge e_{1,5} &\implies e_{2,5} \ .
\end{aligned}
$$

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 $\sum_{i=1}^{n} a_i x_i \leq b$ ($a_i \in \mathbb{Z}$

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

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

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

$$T_A = T_I \rightarrow T_E$$

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.

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

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

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

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

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

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.

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

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

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

- 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*.

Definição (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.

- 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.

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).

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}.

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).

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

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

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

$$(\forall k.k < i \implies a[k] = 0) \wedge a'[i] = 0 \wedge (\forall j \neq i.a'[j] = a[j])$$
$$\wedge\, k_1 \leq i \wedge a'[k_1] \neq 0$$

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

$$(i < i \implies a[i] = 0) \wedge (k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0$$
$$\wedge\, (i \neq i \implies a'[i] = a[i])$$
$$\wedge\, (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0$$

Simplifying, we get

$$(k_1 < i \implies a[k_1] = 0) \wedge a'[i] = 0$$
$$\wedge\, (k_1 \neq i \implies a'[k_1] = a[k_1]) \wedge k_1 \leq i \wedge a'[k_1] \neq 0$$

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.

- 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


Nikolai Bjorner and Leonardo de Moura.
Z3 Theorem Prover.
Rise, Microsft, 2015.


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


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