CS 433 Automated Reasoning 2024

Lecture 20: Theory of arrays

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-04-03

Topic 20.1

Theory of arrays

Theory of arrays

The presence of arrays in programs is ubiquitous.

A solving engine needs to be able to reason over arrays.

Here we present an axiomatization of arrays, which has the following properties.

- arrays are accessible by function symbols $\_[\_]$ and *store*
- $\_[\_]$ and *store* can access an index at a time
- arrays have unbounded length

Commentary: This is a simplified view of arrays. It does not model length of arrays.

Many sorted FOL

FOL is often defined without sorts.

We need many sorted FOL to model arrays, indexes, and values.

In many sorted FOL, a model has a domain that is partitioned for values of each sort.

A term takes value only from its own sort.

Understanding *store* and $\_[\_]$

- $\_[\_]$ returns a value stored in an array at an index

$$\_[\_] : \mathit{Array} \times \mathit{Index} \to \mathit{Value}$$

- *store* places a value at an index in an array and returns a modified array

$$\mathit{store} : \mathit{Array} \times \mathit{Index} \times \mathit{Value} \to \mathit{Array}$$

Array, *Index*, and *Elem* are disjoint parts of the domain of a model.

Commentary: Please leave your programmer's intuition behind. Here arrays are not mutable. A write on array does not modifies the array, but produces a modified copy.

Axiom of theory of arrays (with extensionality)

Let $\mathbf{S}_A \triangleq (\{\_[\_]/2, store/3\}, \emptyset)$. Assuming $=$ is part of FOL syntax.

Definition 20.1

Theory of arrays[†] $\mathcal{T}_A$ *is defined by the following three axioms.*

1. $\forall a \forall i \forall v.\ store(a, i, v)[i] = v$
2. $\forall a \forall i \forall j \forall v.\ i \neq j \Rightarrow store(a, i, v)[j] = a[j]$
3. $\forall a, b.\ \exists i.\ (a \neq b \Rightarrow a[i] \neq b[i])$ *(extensionality axiom)*

The theories that replace the 3rd axiom with some other axiom(s) are called non-extensional theory of arrays

[†] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. (1962) 21-28

Commentary: The axiomatization is simple and powerful. Various solvers use the axioms. The extensionality axiom is considered to be the key source of difficulty, since it introduces a fresh symbol during instantiation.

Models for theory of arrays

A model m contains a set of indexes $Index_m$, a set of values $Value_m$, and a set of arrays $Array_m$. Constants take values from their respective sorts.

Exercise 20.1

Prove $|Array_m| = |Index_m \to Value_m|$ *for model* m.

Example 20.1

Consider the following satisfying model of formula $a[i] = a[j] \land i \neq j$*:*

Let $Index_m = \{1, 2\}$ *and* $Value_m = \{3, 8\}$ *and* $Array_m = \{a_1, a_2, a_3, a_4\}$

- $a_1[1]_m = 3, a_1[2]_m = 3,$
- $a_2[1]_m = 8, a_2[2]_m = 8$
- $a_3[1]_m = 3,\ a_3[2]_m = 8,$
- $a_4[1]_m = 8,\ a_4[2]_m = 3$

$store_m(a_1, 1, 3) = a_1$, $store_m(a_1, 1, 8) = a_4$, $store_m(a_4, 2, 8) =$ ___, ...,

$i_m = 1,\ j_m = 2,\ a_m = a_1$

Decidability

Theory of arrays is undecidable.

However, quantifier-free (QF) fragment is decidable and its complexity is NP.

Example : checking sat in theory of arrays

Example 20.2

Consider the following QF_AX formula: $store(a, i, b[i]) = store(b, i, a[i]) \wedge a \neq b$

Apply axiom 3, $store(a, i, b[i]) = store(b, i, a[i]) \wedge a[j] \neq b[j]$

Due to congruence, $store(a, i, b[i])[j] = store(b, i, a[i])[j] \wedge a[j] \neq b[j]$

case $i = j$*: Due to the axiom 1,* $b[i] = a[i] \wedge a[j] \neq b[j] \leftarrow$ *Contradiction.*

case $i \neq j$*: Due to the axiom 2,* $a[j] = b[j] \wedge a[j] \neq b[j] \leftarrow$ *Contradiction.*

Therefore, the formula is unsat.

Exercise

Exercise 20.2

Show if the following formulas are sat or unsat

1. $a = b \land a[i] \neq b[i]$
2. $a = b \land a[i] \neq b[j]$
3. $store(store(a, j, y), i, x) \neq store(store(a, i, x), j, y) \land i \neq j$

Topic 20.2

A theory solver for $\mathcal{T}_A$

Theory solver for arrays

The key issues of checking sat of conjunction of $\mathcal{T}_A$ literals are

- finding the set of the indices of interest
- finding the witness of disequality

Array solvers lazily/eagerly add instantiations of the axioms for relevant indices

Commentary: An eager solver instantiates axioms all possible relevant ways at pre-solving phase and solves using some EUF solver. A lazy solver instantiates on demand. And, there can be a combination of the two approaches.

A policy of axiom instantiations

Here we present the policy used in Z3 to add the instantiations.

- flattening of clauses
- solve flattened clauses using CDCL($\mathcal{T}_{\text{EUF}}$)
- time to time introduce new clauses due to instantiations.

L. Moura, N. Bjorner, Generalized, Efficient Array Decision Procedures. FMCAD09 (section 2-4)

Commentary: Here is another policy used in another solver:
M. Bofill and R. Nieuwenhuis, A Write-Based Solver for SAT Modulo the Theory of Arrays, FMCAD2008

Flattening

The solver maintains a set of definitions and a set of clauses.

$\_[\_]/\mathit{store}$ terms are replaced by a fresh symbol and the definitions record the replacement.

Example 20.3

Consider clauses: $store(store(a, j, y), i, x) \neq store(store(a, i, x), j, y) \land i \neq j$

Flattened clauses: $u \neq v \land i \neq j$

Definition store: $u \triangleq store(u', i, x), u' \triangleq store(a, j, y),\ v \triangleq store(v', j, y), v' \triangleq store(a, i, y)$

Exercise 20.3

Translate the following in flattened clauses: $store(a, i, b[i]) = store(b, i, a[i]) \land a \neq b$

Commentary: The example is not chosen well. It has only unit clauses.

CDCL($\mathcal{T}_{\mathrm{EUF}}$) on flattened clauses

CDCL($\mathcal{T}_{\mathrm{EUF}}$) is iteratively applied on the flattened clauses as follows.

1. Run CDCL($\mathcal{T}_{\mathrm{EUF}}$) on the flattened clauses
2. If no assignment found then return unsat. Otherwise, $\mathcal{T}_{\mathrm{EUF}}$ has found equivalences that are compatible with the current clauses
3. Add relevant instantiations of array axioms due to the discovery of new equivalent classes
4. If no new instantiations added then return sat. Otherwise, goto 1

Commentary: CDCL assigns truth value to atoms and EUF solver translates the truth values into equivalence classes.

Relevant axiom instantiation

The following rules add new instantiations of the axioms in the clause set.
The instantiated clauses are flattened and added in CDCL($\mathcal{T}_{\text{EUF}}$).

$\sim$ denotes the discovered equivalences in $\mathcal{T}_{\text{EUF}}$

$$\frac{a \triangleq store(b,i,v)}{a[i] = v} \qquad \frac{a \triangleq store(b,i,\_) \quad \_ \triangleq a'[j] \quad a \sim a'}{i = j \lor a[j] = b[j]}$$

$$\frac{a \triangleq store(b,i,\_) \quad \_ \triangleq b'[j] \quad b \sim b'}{i = j \lor a[j] = b[j]} \qquad \frac{a : Array \quad b : Array}{a = b \lor a[k_{a,b}] \neq b[k_{a,b}]}$$

Commentary: Reading the above rules: In the 2nd rule, if a is defined as above, a and a' are equivalent under current assignment, and a' is accessed at j then we instantiate the 2nd axiom involving indexes i and j, and arrays a and b

Soundness and completeness

The solver is sound because it only introduces the instantiations of axioms.

Theorem 20.1

The solver is complete

Proof sketch.

We need to show that only finite and all relevant instantiations are added.
If no conflict is discovered after saturation then we can construct a model. □

Exercise 20.4

Fill the details in the above proof

- *Only finite instantiations are added*
- *Construct a model at saturation*

Optimizations

We may reduce the number of instantiations that are needed to be complete.

Here, we discuss three such optimizations.

- Instantiations for equivalent symbols are redundant
- Instantiate extensionality only if a disequality is discovered in EUF
- Instantiate 2nd axiom only if the concerning index is involved in the final model construction

Redundant Instantiations

If EUF solver has proven $i \sim i'$, $j \sim j'$, $a \sim a'$, and $b \sim b'$ then

$$i = j \vee a[j] = b[j] \qquad \text{and} \qquad i' = j' \vee a'[j'] = b'[j']$$

are mutually redundant instantiations.

We need to instantiate only one of the two.

Similarly, if EUF solver has proven $a \sim a'$, and $b \sim b'$ then

$$a = b \vee a[k_{a,b}] \neq b[k_{a,b}] \qquad \text{and} \qquad a' = b' \vee a'[k_{a',b'}] \neq b'[k_{a',b'}]$$

are mutually redundant instantiations.

Extensionality axiom only for disequalities

We only need to produce evidence that two arrays are disequal only if EUF finds such disequality

$$\frac{a : Array \quad b : Array \quad a \not\sim b}{a = b \vee a[k_{a,b}] \neq b[k_{a,b}]}$$

Restricted instantiation of the 2nd axiom

Definition 20.2

$b \in$ *nonlinear* *if*

1. $b \triangleq store(\_,\_,\_)$ *and there is another* b' *such that* $b \sim b'$ *and* $b' \triangleq store(\_,\_,\_)$
2. $a \triangleq store(b,\_,\_)$ *and* $a \in nonlinear$
3. $a \sim b$ *and* $a \in nonlinear$

We restrict the third instantiation rule as follows.

$$\frac{a \triangleq store(b,i,v) \quad w \triangleq b'[j] \quad b \sim b' \quad b \in nonlinear}{i = j \vee a[j] = b[j]}$$

Theorem 20.2

If $b \notin nonlinear$*, value of index* j *has no effect in the model construction of* a*.*

Topic 20.3

Problems

Swap

Exercise 20.5

Prove the following formula unsatisfiable using the axioms of the theory of arrays

$$store(a, i, b[i]) = store(b, i, a[i]) \wedge a \neq b$$

Prove sorting

Exercise 20.6

Give a model that satisfies the following formula:

$$\forall i, j.\ (i < j \Rightarrow a[i] = b[i]) \Rightarrow \forall i.\ a[i] = b[i+1]$$

Can we also prove?

$$\forall i.\ a[i] = b[i+1] \Rightarrow \forall i, j.\ (i < j \Rightarrow a[i] = b[i])$$

Model generation

Exercise 20.7

Give a model that satisfies the following formula:

$$store(store(b, i_0, b[i_1]), i_1, b[i_0]) = store(store(b, i_1, b[i_1]), i_1, b[i_1])$$

Run Z3

Exercise 20.8
Run Z3 in proof producing mode on the following example:

$$store(store(a, j, y), i, x) \neq store(store(a, i, x), j, y) \wedge i \neq j$$

explain the proof of unsatisfiability produced by Z3.

Note that: In smt-lib format select denotes $\_[\_]$.

Topic 20.4

Extra slides: A decidable fragment of quantified arrays

Decidable fragments

Definition 20.3

An undecidable class often has non-obvious sub-classes that are decidable, which are called decidable fragments.

For example, QF_AX is a decidable fragment of AX.

Finding decidable fragments of various logics is an active area of research.

Now we will present a decidable fragment of AX called "array properties", which allows some restricted form of quantifiers.

For ease of introducing core ideas, the fragment presented here is smaller than the original proposal in

Aaron R. Bradley, Zohar Manna, Henny B. Sipma: What's Decidable About Arrays? VMCAI 2006

Some notation

For formulas/terms F and G, we say

- $G \in F$ if G occurs in F and
- G is QF in F if $G \in F$ and no variable in $FV(G)$ is universally quantified in F

Array properties

Array properties fragment puts the following restrictions.

- *Index* $= \mathbb{Z}$.
- *Value* sort is part of some decidable theory $\mathcal{T}_v$.
- the formulas in the fragment are conjunctions of array properties that are defined next.

Array property

Definition 20.4

An *array property* *is a formula that has the following shape.*

$$\forall \vec{i}.\ (F_I(\vec{i}) \Rightarrow F_V(\vec{i}))$$

- *there are other array, index, and value variables that are free*
- $F_I(\vec{i}) \in guard$

$$\begin{aligned} guard &::= guard \vee guard \mid guard \wedge guard \mid exp \leq exp \mid exp = exp \\ exp &::= i \mid pexp && i \in \vec{i} \\ pexp &::= \mathbb{Z} \mid \mathbb{Z}j \mid pexp + pexp && j \notin \vec{i} \end{aligned}$$

- $F_V(\vec{i})$ *is a QF formula from* $\mathcal{T}_v$. *If* $i \in \vec{i}$ *and* $i \in F_V$ *then* i *only occurs as parameter of some array read and nested accesses are disallowed.*

Example: array properties

Example 20.4

Are the following formulas array properties?

- $\forall i.\ a[i] = b[i]$ ✓
- $\forall i.\ a[i] = b[i+1]$ ✗
- $\forall i.\ a[i] = b[j+1]$ ✓
- $\forall i,j.\ i \leq j \Rightarrow a[i] \leq a[j]$ ✓
- $\forall i,j.\ i \leq j \Rightarrow a[a[i]] \leq a[j]$ ✗
- $\forall i,j.\ i \leq k+1 \Rightarrow a[i] \leq a[j]$ ✓
- $\forall i,j.\ \neg(i \leq k+1) \Rightarrow a[i] \leq a[j]$ ✗
- $\forall i,j.\ i \leq j+1 \Rightarrow a[i] \leq a[j]$ ✗

Decision procedure: notation

For an array property F,

Definition 20.5

The read Set R_F *is* $\{t | \_[t] \in F \wedge t \text{ is QF in } F\}$

Definition 20.6

The bound Set B_F *is* $\{t | (\forall \vec{i}.\ F_I(\vec{i}) \Rightarrow F_V(\vec{i})) \in F \wedge t \bowtie i \in F_I \wedge t \text{ is QF in } F\}$ *where* $\bowtie \in \{\leq, =, \geq\}$.

Definition 20.7

For an array property F, index set $I_F = B_F \cup R_F$

Decision procedure for array properties

1. Replace writes by 1st and 2nd axioms of arrays

$$F[store(a,t,v)] \quad \rightsquigarrow \quad F[b] \wedge b[t] = v \wedge \forall i.\ (i \neq t \Rightarrow a[i] = b[i])$$

   We will call the transformed formula F'.

   Remains in array property fragment

2. Replace universal quantifiers by index sets

$$F'[(\forall \vec{i}.\ F_I(\vec{i}) \Rightarrow F_V(\vec{i}))] \quad \rightsquigarrow \quad F'[\bigwedge_{\vec{t} \in I_{F'}^{len(\vec{i})}} (F_I(\vec{t}) \Rightarrow F_V(\vec{t}))]$$

   We will call the transformed formula F''.

   no universal quantifiers

3. F'' is in QF fragment of $\mathcal{T}_A + \mathcal{T}_{\mathbb{Z}} + \mathcal{T}_v$. We solve it using a decision procedure for the theory combination. We have not covered theory combination yet!!

Exercise 20.9

Extend this procedure for the boolean combinations of array properties.

Example: solving array properties

Example 20.5

Consider:

$x < y \land k + 1 < \ell \land b = store(a, \ell, x) \land c = store(a, k, y) \land$
$\forall i, j.\ (k \leq i \leq j \leq \ell \Rightarrow b[i] \leq b[j]) \land \forall i, j.\ (k \leq i \leq j \leq \ell \Rightarrow c[i] \leq c[j])$

After removing stores:

$x < y \land k + 1 < \ell \land$
$b[\ell] = x \land \forall i.\ (\ell + 1 \leq i \lor i \leq \ell - 1) \Rightarrow b[i] = a[i] \land$
$c[k] = y \land \forall i.\ (k + 1 \leq i \lor i \leq k - 1) \Rightarrow c[i] = a[i] \land$
$\forall i, j.\ (k \leq i \leq j \leq \ell \Rightarrow b[i] \leq b[j]) \land$
$\forall i, j.\ (k \leq i \leq j \leq \ell \Rightarrow c[i] \leq c[j])$

Exercise 20.10

The index set for the above formula includes expression $k - 1$. Instantiate the last quantified formula for term $k - 1$.

Commentary: Removing stores may introduce new arrays. The above example is simple enough and we need not introduce new arrays.

Example: solving array properties(contd.)

Index set $I = \{k-1, k, k+1, \ell-1, \ell, \ell+1\}$
We instantiate each universal quantifier 6 times.
Therefore, 84 quantifier-free clauses are added.

Let us consider only the following instantiations of the quantifiers:
$x < y \wedge k+1 < \ell \wedge b[\ell] = x \wedge c[k] = y \wedge$
$(\ell+1 \leq k+1 \vee k+1 \leq \ell-1) \Rightarrow b[k+1] = a[k+1] \wedge$
$(k+1 \leq k+1 \vee k+1 \leq k-1) \Rightarrow c[k+1] = a[k+1] \wedge$
$k \leq k \leq k+1 \leq \ell \Rightarrow c[k] \leq c[k+1] \wedge$
$k \leq k+1 \leq \ell \leq \ell \Rightarrow b[k+1] \leq b[\ell] \wedge$ ....(many more)

Since all the above mentioned guards are true,
$x < y = c[k] \leq c[k+1] = a[k+1] = b[k+1] \leq b[\ell] = x$
Contradiction.

Why are finite instantiations sufficient for checking sat of $\forall$ quantifiers?

Correctness

Theorem 20.3
If F is sat iff F' is sat

Proof.
This step only explicates theory axioms. Trivially holds. □

Theorem 20.4
If F' is sat iff F'' is sat

Proof.
Since F'' is finite instantiations of F', if F'' is unsat then F' is unsat.

Now we show that if $m'' \models F''$ then we can construct a model m' for F'.

Let $I_{F'} = \{t^1, ..., t^\ell\}$. Wlog, we assume $t^1_{m''} \leq ... \leq t^\ell_{m''}$. ...

Correctness (contd.)

Proof(contd.)

Observation:

m'' assigns values to all non-array variables of F'.

In arrays, m'' assigns values only at indexes $I_{F'}$.(Why?)

Constructing m':

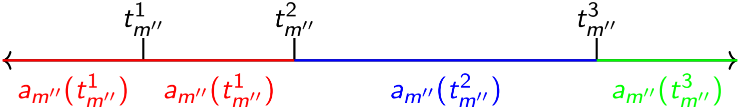
We copy assignment of non-array variables from m'' to m'.

Let a be an array appearing in F'. We construct $a_{m'}$ as follows.

For each $j \in \mathbb{Z}$,

$$a_{m'}(j) \triangleq a_{m''}(t^k_{m''})$$

where $k = \max\{1\} \cup \{j | t^j_{m''} \leq j\}$.



...

Correctness (contd.)

Proof(contd.)

Claim: $m' \models F'$

Consider $\forall \vec{i}.\ F_I(\vec{i}) \Rightarrow F_V(\vec{i}) \in F'$.

$\vec{v}$ is inside a hyper-cube defined by corners $\vec{u}$ and $\vec{w}$

Let $\vec{v} \in \mathbb{Z}^n$, where $n = len(i)$.
Choose $\vec{u} \triangleq (t^{j_1}_{m''}, .., t^{j_n}_{m''})$ and $\vec{w} \triangleq (t^{j_1+1}_{m''}, .., t^{j_n+1}_{m''})$ such that $\vec{u} \leq \vec{v} < \vec{w}$.
Since $m'' \models F''$, $m''[\vec{i} \to \vec{u}] \models F_I(\vec{i}) \Rightarrow F_V(\vec{i})$.

Case $m''[\vec{i} \to \vec{u}] \models F_I(\vec{i})$:

Therefore, $m''[\vec{i} \to \vec{u}] \models F_V(\vec{i})$.
Therefore, $m''[\vec{i} \to \vec{v}] \models F_V(\vec{i})$. (Why?)
Therefore, $m''[\vec{i} \to \vec{v}] \models F_I(\vec{i}) \Rightarrow F_V(\vec{i})$.

...

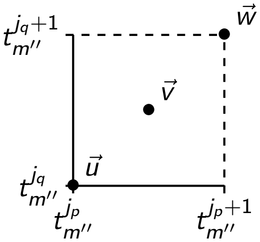
Commentary: This proof divides quantified space into finite parts and chooses a representative such that if it satisfies the formula then all in the partition satisfy the formula.

Correctness (contd.)

Case $m''[\vec{i} \to \vec{u}] \not\models F_I(\vec{i})$:

For $i_p, i_q \in \vec{i}$, there are three kinds of atoms in F_I.

- $i_p \leq t^r$
- $t^r \leq i_p$
- $i_p \leq i_q$



If an atom is false in $m''[\vec{i} \to \vec{u}]$ then it is false in $m''[\vec{i} \to \vec{v}]$.(Why?)
Since F_I is positive boolean combination of the atoms, $m''[\vec{i} \to \vec{v}] \not\models F_I(\vec{i})$.
Therefore, $m''[\vec{i} \to \vec{v}] \models F_I(\vec{i}) \Rightarrow F_V(\vec{i})$.

Exercise 20.11

Some ranges of $\vec{i}$ are missing in the above argument. Complete the proof.

Commentary: Note that if any atom is true at $\vec{u}$ then it can become false at $\vec{v}$.

Finite width

There are no true integers in computers.

Numbers are stored in fixed-width bitvectors.

Example 20.6

$$(x - y \geq 0) \quad \not\equiv \quad x \geq y$$

We need precise reasoning for fixed-with bitvectors.

Topic 20.5

Extra slides: Theory of bitvectors

Sorts and variables

- Bitvector sorts

```
(_ BitVec num)
```

- Declaration of bitvectors

```
(declare-fun x3 () (_ BitVec 32))
```

- Declaring bitvector constants

```
(bv 1 4)
(bv #b0010001 7)
```

Operators on bitvectors

- Bitwise operators
- Vector operators
- Arithmetic operators

Arithmetic operations preserve vector length.

`concat` and `extract` operators are used to change bit lengths.

Bitwise operators

- `bvand: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
  - It takes two vectors of same length, and returns a vector that is bitwise and of inputs.
- similarly `bvor`, `bvxor`, `bvnot` are defined

Vector operators

- `concat: ((_ BitVec m) (_ BitVec n)) (_ BitVec m+n)`
- `(_ extract i j): ((_ BitVec m)) (_ BitVec i-j)`
- `(_ rotate_left i): ((_ BitVec n)) (_ BitVec n)`
- `(_ rotate_right i): ((_ BitVec n)) (_ BitVec n)`
- `(_ bvshl i): ((_ BitVec n)) (_ BitVec n)` shift left

Exercise 20.12

- `(concat 101 10)` =
- `((_ extract 1 4) 10110)` =
- `((_ rotate_left 2) 10110)` =
- `((_ rotate_right 2) 10110)` =
- `((_ bvshl 2) 10110)` =

Signed/unsigned interpration of bitvectors

Bitvectors as number. Let b be a bitvector of length n.

There are two well known ways to interpret b.

- Unsigned number:
$$u\_num(b) := 2^{n-1}b[n-1] + \cdots + 2^0 b[0]$$

- Singed number:
$$s\_num(b) := -2^{n-1}b[n-1] + 2^{n-2}b[n-2] + \cdots + 2^0 b[0]$$

In signed number, the highest bit indicate sign.

Sign aware vector operators

There are two kinds of shift right

- `bvlshr`
  - Pads 0 while shifting
- `bvashr`
  - Padded bits are copy of the highest bit

Two kinds of extension on the left size of bitvector

- `zero_extend`
- `sign_extend`

Exercise 20.13

- `((_ bvlshr 1) 10110)` =
- `((_ bvashr 2) 10110)` =
- `((_ zero_extend 2) 10110)` =
- `((_ sign_extend 2) 10110)` =

Exercise 20.14

a. Prove `sign_extend` preserves s_num value, b. Prove `zero_extend` preserves u_num value

Arithmetic Operators

Addition, subtraction, and multiplication need not know the interpretation

- `bvadd: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvmul: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvsub: ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

Definition 20.8

Let a and b be bitvectors of size n.

`u_num( (bvadd a b) ) = (u_num(a) + u_num(b)) mod` 2^n
`u_num( (bvsub a b) ) = (u_num(a) - u_num(b)) mod` 2^n
`u_num( (bvmul a b) ) = (u_num(a) * u_num(b)) mod` 2^n

Theorem 20.5

Let a and b be bitvectors of size n.

`s_num( (bvadd a b) ) mod` $2^n =$ `(s_num(a) + s_num(b)) mod` 2^n
`s_num( (bvsub a b) ) mod` $2^n =$ `(s_num(a) - s_num(b)) mod` 2^n
`s_num( (bvmul a b) ) mod` $2^n =$ `(s_num(a) * s_num(b)) mod` 2^n

Arithmetic Operators II

Computing negative of a variable and division needs to be aware of the interpretation

- `bvneg:  ((_ BitVec n)) (_ BitVec n)`

- `bvsrem:  ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvsdiv:  ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

- `bvurem:  ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`
- `bvudiv:  ((_ BitVec n) (_ BitVec n)) (_ BitVec n)`

Signed Arithmetic comparators

- `bvslt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsgt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsle : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvsge : ((_ BitVec n) (_ BitVec n)) Bool`

Definition 20.9

Let a and b be bitvectors of size n.

$$\texttt{(bvslt a b)} \Leftrightarrow \texttt{s\_num(a)} < \texttt{s\_num(b)}$$
$$\texttt{(bvsgt a b)} \Leftrightarrow \texttt{s\_num(a)} > \texttt{s\_num(b)}$$
$$\texttt{(bvsle a b)} \Leftrightarrow \texttt{s\_num(a)} \leq \texttt{s\_num(b)}$$
$$\texttt{(bvsge a b)} \Leftrightarrow \texttt{s\_num(a)} \geq \texttt{s\_num(b)}$$

Unsigned Arithmetic comparators

- `bvult : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvugt : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvule : ((_ BitVec n) (_ BitVec n)) Bool`
- `bvuge : ((_ BitVec n) (_ BitVec n)) Bool`

Definition 20.10

Let a and b be bitvectors of size n.

$$(\texttt{bvult a b}) \Leftrightarrow \texttt{u\_num(a)} < \texttt{u\_num(b)}$$
$$(\texttt{bvugt a b}) \Leftrightarrow \texttt{u\_num(a)} > \texttt{u\_num(b)}$$
$$(\texttt{bvule a b}) \Leftrightarrow \texttt{u\_num(a)} \leq \texttt{u\_num(b)}$$
$$(\texttt{bvuge a b}) \Leftrightarrow \texttt{u\_num(a)} \geq \texttt{u\_num(b)}$$

Topic 20.6

Extra slides: Solving bitvector formulas

BV theory solving

bitvector theory solvers work in the following two stages

- term rewriting
- bit blasting (SAT encoding)

Term rewriting to deal with high level structure and
bit blasting for unstructured boolean reasoning.

Term rewriting

Here are some rewriting phases that are helpful

- bvToBool: lift boolean statements to bitvector statements
  - For example:
    $(x[i:i]) \wedge ite(c, x[1], 0[1]) = 1[1]$ is translated to $(x[i:i] = 1[1]) \wedge ite(c, x[1] = 1[1], \bot)$
- ackermannize
  - For example: $\phi(f(x), f(y))$ is translated to $\phi(x', y') \wedge (x = y \Rightarrow x' = y')$
- algebraic pattern detection
  - For example: $x * x + x$ translated to $x * (x + 1)$

Refactoring isomorphic circuits

We may use information learned by term rewriting in clause learning

For example, refactoring isomorphic circuits

Consider the following formula

$$(x_0 = 2 * y_0 + y_1 \vee x_0 = 2 * y_1 + y_2 \vee x_0 = 2 * y_2 + y_0) \wedge \phi$$

We may observe that x_0 is assigned by expressions.

We recognize the pattern and rewrite the formula

$$\forall x, x'.\ f(x, x') = 2 * x + x' \wedge (x_0 = f(y_0, y_1) \vee x_0 = f(y_1, y_2) \vee x_0 = f(y_2, y_0)) \wedge \phi$$

How is it helpful?

Refactoring isomorphic circuits(contd.)

This information can be used multiple ways

- Learned clause reuse
  - Each application of f will produce isomorphic clauses after bit blasting.
  - A clause learned one set of clause can be translated to another learned clause for the isomorphic clause sets.
  - For example, if the clause learning detects first bit of x_0 and y_1 are equal since $x_0 = f(y_0, y_1)$. We may similar clauses to the other applications of f.
  - Minimize the encoding: the rewritten formula will have far less number of clauses

Topic 20.7

Extra slides: Bit blasting

Bit blasting: Translating to clauses

If high-level reasoning does not result in any answer.

In bit blasting, we convert every BV arithmetic expressions to Boolean clauses

Let us see a few such translations

Translating comparison

Consider `(bvult a b)`, where a and b are bitvectors of size n.

We can encode the bitvector formula into the following propositional formula.

$$cmp(a, b, n) := (\neg a[n-1] \wedge b[n-1]) \vee (a[n-1] = b[n-1] \wedge cmp(a, b, n-1))$$
$$cmp(a, b, 0) := \bot$$

Exercise 20.15

a. encode `(bvule a b)`
b. encode `(bvugt a b)`
c. encode `(bvslt a b)`

Translating addition

Consider `sum = (bvadd a b)`, where a and b are bitvectors of size n.

We can encode addition as follows

$carry[-1] = \bot$

$carry[i] := (a[i] \wedge b[i]) \vee ((a[i] \oplus b[i]) \wedge carry[i-1])$

$sum[i] \Leftrightarrow a[i] \oplus b[i] \oplus carry[i-1]$

Translating multiplication

Consider `res = (bvmul a b)`, where a and b are bitvectors of size n.

We can encode the multiplication as follows

$mul(a, b, -1) := b$

$mul(a, b, s) :=$ `(bvadd mul(a,b,s-1) (ite b[s] (bvshl a s) 0) )`

Exercise 20.16

What is the multiplier encoding in Z3, Boolector, and CBMC?

Translating division

Consider `d = (bvudiv a b)` and `r = (bvurem a b)`, where a and b are bitvectors of size n.

We can encode the unsigned division as follows

$$b \neq 0 \Rightarrow (mul(d, b, n) + r = a) \wedge r < b$$

Exercise 20.17

encode `d = (bvsdiv a b)` and `r = (bvsrem a b)`

Topic 20.8

Extra slides: Lazy bit blasting

Eagar bit blasting may hurt!

There are no ways to encode multiplications efficiently.

Example 20.7

The following formula is unsatisfiable for a simple reason.

$$a = b * c \land a < b \land a > b$$

But, the eager bit blasting blows up the encoding due to multiplication and may loose structure.

Bit blast on demand!

1. We treat bitvector operators as uninterpreted functions/relations, with other functional proprieties, e. g., `bvslt` is antisymmetric.

2. If the formula remains satisfiable, we find the operations that are violated by the current assignment.

3. If no such violation found, the formula is satisfiable.

4. We add bit blasted encoding of the violated operations and goto 2.

Commentary: We need not add encodings of all the violated operations immediately. As long as we add at least one in each iteration, the procedure is sound and terminates.

Topic 20.9

Extra slides: Modular arithmetic

Can we use linear arithmetic solver?

If there are not too many "bitwise operations" in input formulas,

we can translate the formulas into integer linear arithmetic constraints.

We need to be aware of the modular operations, also called overflows.

Encoding into linear arithmetic

We translate the bitvectors to integer constraints in the following steps.

1. Remove bitwise operations
2. Remove division
3. Remove constant multiplications
4. Replace additions with overflow aware additions

Naturally, if the input in not linear, we can not encode in the integer linear arithmetic.

Remove bitwise operations

- Translating `(bvnot b)`

$$-b+1$$

- Translating `x = (bvand b 1)`

$$(b = 2y + x \wedge y < 2^{n-1} \wedge 0 \leq x \leq 1)$$

Exercise 20.18

a. Provide encoding for `bvshl`.
b. Provide encoding for `bvand` with arbitrary constants.
c. Provide encoding for `bvor` with arbitrary constants.

Remove division by constants

Translating `x = (bvudiv b k)`

$$b = x * k$$

Remove bitvector multiplication by constants

- For small constants repeat addition
- For large constants `(bvmul b c)` translates to

$$b * c - x2^n$$

with supporting constraints $b * c - x2^n < 2^n \wedge x < c - 1$

Removing bitvector addition

We replace addition `(bvadd s t)` as follows

$$ite(s + t < 2^n, s + t, s + t - 2^n)$$

Such a translation is not always helpful! It may end up introducing a large number of case splits.

End of Lecture 20