Weakly Equivalent Arrays

Jürgen Christ Jochen Hoenicke

University of Freiburg

July 17th, 2014

Motivation

```
var a : [int]int;
procedure swap(i, j : int)
ensures a[i] = old(a[j])
     && a[j] = old(a[i])
{
  a[i] = a[i] + a[j];
  a[j] = a[i] - a[j];
  a[i] = a[i] - a[j];
}
```

$$
\begin{aligned}
& t_1 = \mathsf{select}(a_0, j_0) \\
\wedge\ & a_1 = \mathsf{store}(a_0, i_0, \mathsf{select}(a_0, i_0) + \mathsf{select}(a_0, j_0)) \\
\wedge\ & a_2 = \mathsf{store}(a_1, i_0, \mathsf{select}(a_1, i_0) - \mathsf{select}(a_1, j_0)) \\
\wedge\ & a_3 = \mathsf{store}(a_2, i_0, \mathsf{select}(a_2, i_0) - \mathsf{select}(a_2, j_0)) \\
\wedge\ & \neg(\mathsf{select}(a_3, i_0) = \mathsf{select}(a_0, j_0) \\
& \qquad \wedge \mathsf{select}(a_3, j_0) = \mathsf{select}(a_0, i_0))
\end{aligned}
$$

We want to

- check satisfiability of this formula: SAT!
- compute a model:

$$
\begin{aligned}
i_0 = j_0 &= 1 \\
a_0 &= \mathsf{store}(\mathsf{store}(@0, 2, 1), 1, -1) \\
a_1 &= \mathsf{store}(\mathsf{store}(@0, 2, 1), 1, -2) \\
a_2 = a_3 &= \mathsf{store}(\mathsf{store}(@0, 2, 1), 1, 0)
\end{aligned}
$$

Section 1

SMT-Theory of Arrays

Arrays are basically (functional) hash maps

The Theory of Arrays defines

- a sort Array *Index Elem* for arbitrary sorts *Index* and *Elem*.
- two (polymorphic) functions
  - select : Array $I\ E \times I \to E$
    select(a, i) reads the element stored at index i in array a and returns it.
  - store : Array $I\ E \times I \times E \to$ Array $I\ E$
    store(a, i, v) stores the element v into array a at index i and returns the new arrays (a itself is unchanged).

Axioms of the Theory of Arrays

The Theory of Arrays is defined by the following three axioms.

- Select-over-Store-1 (what you store is what you get)

$$\forall abijv.i = j \wedge b = \text{store}(a, i, v) \rightarrow \text{select}(b, j) = v$$

- Select-over-Store-2 (store does not change other entries)

$$\forall abijv.i \neq j \wedge b = \text{store}(a, i, v) \rightarrow \text{select}(b, j) = \text{select}(a, j)$$

- Extensionality:

$$\forall ab.(\forall i.\text{select}(a, i) = \text{select}(b, i)) \rightarrow a = b$$

Solving by Instantiation of Axioms

Existing Decision Procedure for Arrays:

- input: quantifier free formula ϕ over the Theory of Arrays.
- output: equisatisfiable quantifier free formula ϕ' over the Theory of Indices and Elements.
- Instantiate each axiom $\forall abijv.\psi(a, b, i, j, v)$ from the previous slide with each combination of sub-term t_a, t_b, t_i, t_j, t_v of ϕ with the corresponding sort.
- Add all instantiated formula $\psi(t_a, t_b, t_i, t_j, t_v)$ to ϕ to obtain ϕ'

Problems of the existing approach

UNI FREIBURG

- Requires many ($O(n^5)$) instantiations, most of them not relevant.
  $\rightarrow$ lazily add instances when needed (how?)
- Bad for interpolation: instances mix terms from different parts.

Christ, H., Nutz: Proof Tree Preserving Interpolation TACAS 2013

presents an algorithm that handles mixed literals $a = b$.
However, instantiation produces mixed terms

$$\text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)$$

which is not supported by the interpolation algorithm.

Decision-Procedure for Arrays

UNI FREIBURG

We developed a new algorithm that

- does not create new select/store-terms,
- creates array lemmas lazily on the fly,
- fits nicely into the Nelson–Oppen scheme,
- DPLL and CDCL-friendly,
- produces models for satisfiable formulae and proofs for unsatisfiable formulae.

Section 2

Nelson–Oppen Combination

Nelson–Oppen Theory Combination

Idea: Separate Decision Procedure for every Theory
Theories share only equality
Propagate implied equalities between Theories.

Example:

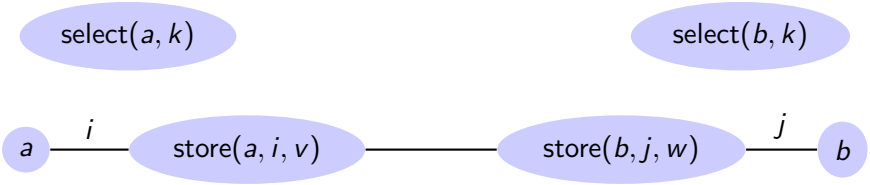
Integer Theory:	$i \geq j \wedge j \geq i + z \wedge z \geq 0$
Array Theory:	$b = \text{store}(a, i, v) \wedge \text{select}(b, j) = w$
Function Theory:	$f(v) \neq f(w)$

1. Integer Theory propagates $i = j$.
2. Array Theory propagates $w = v$.
3. Function Theory detects conflict.

Array Theory

Algorithm for Array Theory.
Input: Equivalence classes for shared variables.
Output: New lemmas that are valid in the array theory.

Example:
$V = \{a, b, i, j, k, v, w, \text{store}(a, i, v), \text{store}(b, j, w), \text{select}(a, k), \text{select}(b, k)\}$
Equalities: $\text{store}(a, i, v) = \text{store}(b, j, w)$



New Lemma: $\text{store}(a, i, v) = \text{store}(b, j, w) \wedge k \neq i \wedge k \neq j \Rightarrow$
$\text{select}(a, k) = \text{select}(b, k)$.

We need a generalized read-over-store axiom.

Section 3

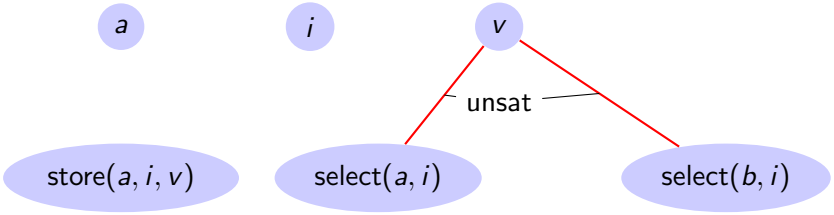
(Weak) Equivalence Graphs

Equivalence Graphs

The theory of equality is usually decided by equivalence graphs
(more precisely: Union-Find-Structures).
Given a conjunction of (dis-)equality literals.

- Create a vertex for every (non-Boolean) sub-term of the input formula.
- Create an edge for every equality literal.
- Unsat iff there is a disequality where the vertices are connected.

Example: $\text{select}(b, i) = v \wedge \text{select}(a, i) = v \wedge \text{select}(b, i) \neq \text{select}(a, i)$

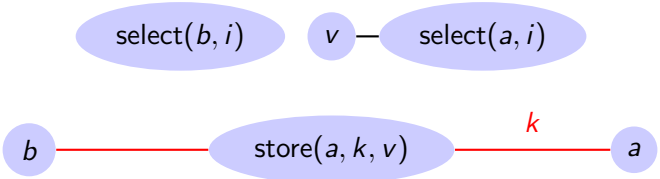


Weak Equivalence Graphs

Two arrays are weakly equivalent, if they are equal for almost all indices.

1. Create the equivalence graph for the arrays as before.
2. Add a weak edge $a \overset{k}{---} \text{store}(a, k, v)$ for every term $\text{store}(a, k, v)$ occuring in the graph.

Example: $b = \text{store}(a, k, v) \wedge \text{select}(b, i) \neq v \wedge \text{select}(a, i) = v$



Notations:

- Weak equivalent: $b \overset{\phi}{\leftrightarrow} a$ iff there is a path ϕ.
- Weak equivalent modulo i: $b \approx_i a$ iff $b \overset{\phi}{\leftrightarrow} a$ with $i \neq k$ for all $k \in \phi$.

Read-over-Weakeq Rule

$$\frac{a \approx_i b \qquad i = j \qquad \mathsf{select}(a, i), \mathsf{select}(b, j) \in V}{i = j \wedge \mathit{Cond}(a \approx_i b) \rightarrow \mathsf{select}(a, i) = \mathsf{select}(b, j)}$$

where

$$\mathit{Cond}(a \approx_i b) := \bigwedge_{\mathrm{edge} \in \phi} \mathit{Cond}_i(a \text{ edge } b)$$

$$\mathit{Cond}_i(a \text{ — } b) := a = b$$

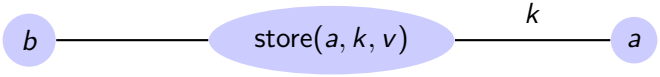
$$\mathit{Cond}_i(a \xrightarrow{k} \mathsf{store}(a, k, v)) := i \neq k$$

For Select-Over-Store-1 we apply the following rule at the beginning

$$\frac{\mathsf{store}(a, k, v) \in V}{\mathsf{select}(\mathsf{store}(a, k, v), k) = v}$$

Example

$b = \text{store}(a, k, v) \wedge \text{select}(b, i) \neq v \wedge \text{select}(a, i) = v$



$a \approx_i b$ with the path $b = \text{store}(a, k, v) \xrightarrow{k} a$.
V contains shared variables for $\text{select}(b, i), \text{select}(a, i)$.

Derived Lemma $b = \text{store}(a, k, v) \wedge i \neq k \rightarrow \text{select}(b, i) = \text{select}(a, i)$.

Soundness and Completeness

- Soundness holds.
  (Rule read-over-weakeq is derivable from select-over-store).
- Complete for Arrays without Extensionality.
  A consistent model for the arrays can be created.

Section 4

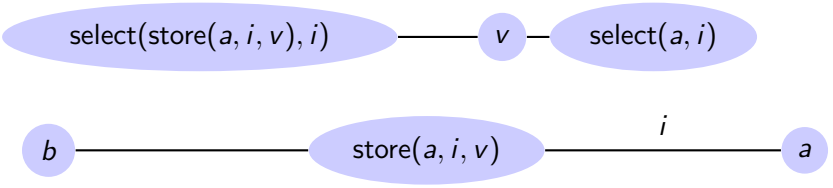
Extensionality

Extended Weak Equivalence Graphs

For extensionality we need the even weaker relation "weak congruence"

$$a \sim_i b \iff \begin{array}{l} a \approx_i b \vee \\ a \approx_i a' \wedge i = j \wedge \mathrm{select}(a', j) = \mathrm{select}(b', k) \wedge k = i \wedge b' \approx_i b' \end{array}$$

Example: $b = \mathrm{store}(a, i, v) \wedge \mathrm{select}(a, i) = v \wedge \mathrm{select}(\mathrm{store}(a, i, v), i) = v$



For $i \neq j$ we get $a \sim_j b$ from $a \approx_j b$.

For i we have $a \approx_i a \wedge i = i \wedge \mathrm{select}(a, i) = v = \mathrm{select}(\mathrm{store}(a, i, v), i) \wedge i = i \wedge \mathrm{store}(a, i, v) \approx_i b$. Hence $a \sim_i b$.

Extensionality

Extended proof rule for extensionality:

$$\frac{a \overset{\phi}{\Leftrightarrow} b, \forall i \in Stores(\phi).a \sim_i b}{Cond(\phi) \wedge \bigwedge_{i \in Stores(\phi)} Cond(a \sim_i b) \rightarrow a = b}$$

where

$$Cond(a \sim_i b) := \begin{pmatrix} Cond(a \approx_i a') \wedge \\ i = j \wedge a'[j] = b'[k] \wedge k = i \\ \wedge\ Cond(b' \approx_i b) \end{pmatrix}$$

Soundness and Completeness

UNI FREIBURG

Theorem

The calculus with the extended read-over-write axiom and the extended extensionality axiom is sound and complete.

Our assumptions:

- Index sort is stably infinite.
- Element sort contains at least two elements (Booleans are allowed).

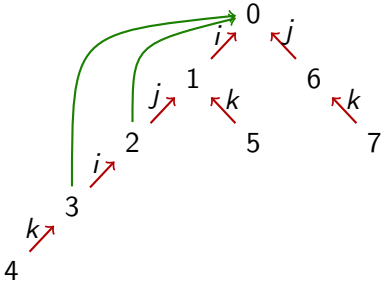
$\rightarrow$ Array sort is stably infinite.

Section 5

Implementation in SMTInterpol

Data Structure for Weak Equivalence

How can we efficiently represent all weak equivalence classes?



- Nodes are (strong) equivalence classes on shared variables.
- Primary edges represent the edges introduced for stores.
  They build a spanning tree pointing to the representative of the weak equivalence class
- Secondary edges used when primary edge has the wrong index.

Caveats

UNI FREIBURG

- Building the weak equivalence DAG is worst case $O(n^3)$.
  Much worse than $O(n \log(n))$ for congruence closure.
- DAG cannot be built incrementally
  (a new equality literal on indices may invalidate the whole DAG).
- Still performance is "good enough" to run before every decision point.

Evaluation

We participated in the SMTCOMP 2014. We support three array divisions (quantifier-free and without bit-vectors) containing 1042 benchmarks.

Solver	Solved/Total	CPU time
Yices	1042/1042	92.67
SMTInterpol	1042/1042	2663.01
[Z3]	1042/1042	5276.94
[MathSAT]	1023/1042	8626.92
CVC4	1022/1042	11679.75

Notes:

- The solver is implemented in Java and the start-up overhead (for JIT-compiling) is not negligable.
- SMTInterpol has no specialized pre-processor techniques.

Future Work

- Interpolation support for array lemmas.
  The proofs can be handled by our proof tree preserving interpolation procedure. We "just" need to interpolate the theory lemmas.
- Performance issues:
  - Better support for incremental solving in weak equivalence DAGs.
  - Is there a way to avoid $O(n^3)$?

http://ultimate.informatik.uni-freiburg.de/smtinterpol/