Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 8:
Procedures for First-Order Theories, Part 2

Matt Fredrikson
mfredrik@cs.cmu.edu

October 17, 2016
We will make things simpler by **removing predicate symbols**

**Signature:**

\[ \Sigma_E : \{=, a, b, c, \ldots, f, g, h, \ldots \} \]

**Axioms:**

1. **Reflexivity:** \( \forall x. x = x \)
2. **Symmetry:** \( \forall x, y. x = y \rightarrow y = x \)
3. **Transitivity:** \( \forall x, y, z. x = y \land y = z \rightarrow x = z \)
4. **Function congruence:** \( \forall x, y. (\land_{i=1}^{n} x_i = y_i) \rightarrow f(x) = f(y) \)

This is the **Theory of Equality and Uninterpreted Functions (EUF)**
Decision Procedure for $T_E$-Satisfiability

Given a $T_E$-formula

$$F : s_1 = t_1 \land \cdots \land s_m = t_m \land s_{m+1} \neq t_{m+1} \land \cdots \land s_n \neq t_n$$

with subterm set $S_F$:

1. Construct the DAG for $S_F$

2. For $i \in \{1, \ldots, m\}$, merge $s_i$ and $t_i$

3. If $\text{find}(s_i) = \text{find}(t_i)$ for an $i \in \{m + 1, \ldots, n\}$, then $\text{unsat}$

4. If $\text{find}(s_i) \neq \text{find}(t_i)$ for all $i \in \{m + 1, \ldots, n\}$, then $\text{sat}$
Signature:

\[ \Sigma_A : \{ =, [\cdot], \langle \cdot \triangleleft \cdot \rangle \} \]

- \( a[i] \) is a binary function denoting read of \( a \) at index \( i \)
- \( a \langle i \triangleleft v \rangle \) is a ternary function denoting write of value \( v \) into \( a \) at index \( i \)

We’ll see how to decide the quantifier-free, conjunctive fragment

- Is this expressive?
- Can only talk about individual elements, not entire arrays
- Afterwards, we’ll cover a quantified fragment
$T_A$: Axioms

$T_A$ has the equality axioms reflexivity, symmetry, and transitivity

Along with three that are specific to arrays:

1. **Array congruence:** $\forall a, i, j. i = j \rightarrow a[i] = a[j]

2. **Read-over-write 1:** $\forall a, v, i, j. i = j \rightarrow a\langle i < v\rangle[j] = v$

3. **Read-over-write 2:** $\forall a, v, i, j. i \neq j \rightarrow a\langle i < v\rangle[j] = a[j]$
Basic Idea: We’ll reduce this to deciding $T_E$

- If a $T_A$-formula has no writes, then reads can be viewed as uninterpreted function terms
- If there is a write, it must occur in the context of a read. Why?
- So all writes occur in read-over-write terms $a\langle i<v\rangle[j]$  
- We apply the read-over-write axioms to decompose these terms into simpler ones
- Then we use our $T_E$ solver
Deciding Theory of Arrays, In Detail

Given $T_A$-formula $F$, follow these steps recursively:

If $F$ doesn’t contain any write terms, do the following:
1. Associate each array variable $a$ with a fresh function symbol $f_a$
2. Replace each read term $a[i]$ with $f_a(i)$
3. Decide and return the $T_E$ satisfiability of the resulting formula

Otherwise, select a term $a\langle i \triangleleft v \rangle[j]$, and split into cases:
1. By (read-over-write 1), replace $F[a\langle i \triangleleft v \rangle[j]]$ with $F_1 : F[v] \land i = j$.
2. By (read-over-write 2), repl. $F[a\langle i \triangleleft v \rangle[j]]$ with $F_2 : F[a[j]] \land i \neq j$.
3. Recurse on $F_1$ and $F_2$. If both are unsat, then return unsat.
4. If either is sat, then return sat
$F : \ i_1 = j \land i_1 \neq i_2 \land \ a[j] = v_1 \land a\langle i_1 < v_1 \rangle \langle i_2 < v_2 \rangle [j] \neq a[j]$

$F$ has a write term, so select a read-over-write term to deconstruct:

$$a\langle i_1 < v_1 \rangle \langle i_2 < v_2 \rangle [j]$$

According to (read-over-write 1), assume $i_2 = j$ and recurse on:

$$F_1 : \ i_1 = j \land i_1 \neq i_2 \land a[j] = v_1 \land v_2 \neq a[j] \land i_2 = j$$

This doesn’t have any write terms, so build a $T_E$-formula:

$$F'_1 : \ i_1 = j \land i_1 \neq i_2 \land f_a(j) = v_1 \land v_2 \neq f_a(j) \land i_2 = j$$

This is unsatisfiable, so let’s move on to the next case.
Example

\[ F : i_1 = j \wedge i_1 \neq i_2 \wedge a[j] = v_1 \wedge a\langle i_1 < v_1 \rangle \langle i_2 < v_2 \rangle [j] \neq a[j] \]

According to (read-over-write 2), assume \( i_2 \neq j \) and recurse on:

\[ F_2 : i_1 = j \wedge i_1 \neq i_2 \wedge a[j] = v_1 \wedge a\langle i_1 < v_1 \rangle [j] \neq a[j] \wedge i_2 \neq j \]

This has a write term, so apply (read-over-write 1) and assume \( i_1 = j \):

\[ F_3 : i_1 = j \wedge i_1 \neq i_2 \wedge a[j] = v_1 \wedge v_1 \neq a[j] \wedge i_2 \neq j \]

This is unsatisfiable, so (read-over-write 2) and assume \( i_1 \neq j \):

\[ F_3 : i_1 = j \wedge i_1 \neq i_2 \wedge a[j] = v_1 \wedge a[j] \neq a[j] \wedge i_1 \neq j \]

Now all branches have been tried, and we conclude that \( F \) is \( T_A\text{-unsat} \).
Bringing Back Quantifiers

Reasoning about quantifier-free $T_A$ reduces to $T_E$

This is great—we know how to decide $T_E$ efficiently

But we’re very limited in what we can say about arrays

For example, we can’t say anything equivalent to:

$$a[i] = v \rightarrow a\langle i \triangleleft v \rangle = a$$

Now, we add quantifiers to define the **array property fragment**
Array Properties

An **array property** is a $T_A$-formula of the form:

$$\forall i. F[i] \rightarrow G[i]$$

where \( i \) is bold to denote a list of variables.

Some terminology:

- \( F[i] \) is the **index guard**
- \( G[i] \) is the **value constraint**

In the value constraint:

- \( \forall \)-quantified variables in \( G[i] \) can only appear in a read \( a[i] \)
- \( \forall \)-quantified variables cannot occur in nested reads, e.g., \( a[b[i]] \)
We restrict how the index guard may be constructed:

\[
\text{iguard} ::= \text{iguard} \land \text{iguard} \mid \text{iguard} \lor \text{iguard} \mid \text{atom}
\]

\[
\text{atom} ::= \text{var} = \text{var} \mid \text{evar} \neq \text{var} \mid \text{var} \neq \text{evar} \mid \top
\]

\[
\text{var} ::= \text{evar} \mid \text{uvar}
\]

- \text{uvar} is a $\forall$-quantified variable
- \text{evar} is an unquantified free variable

The array property fragment consists of Boolean combinations of:

- Quantifier-free $T_A$ formulas
- Array properties
Example

Is the following formula in the array property fragment?

$$\forall i. i \neq a[k] \rightarrow a[i] = a[k]$$

**No.** $i \neq a[k]$ is not a legal index guard as $a[k]$ is not a variable

Can we find an equisatisfiable formula in the fragment?

$$a[k] = v \land \forall i. i \neq v \rightarrow a[i] = a[k]$$

What about this formula?

$$\forall i. i \neq a[i] \rightarrow a[i] = a[k]$$

$i$ is bound by the quantifier

We can’t move $a[i]$ into a quantifier-free formula
Example

How can we express:

\textit{All but a finite number of elements of $a$ satisfy $F$}

\[\forall i. \left( \bigwedge_{k=1}^{n} i \neq j_k \right) \rightarrow F[a[i]]\]

\textit{Updating element $i$ of $a$ only changes $a[i]$}

\[\forall j. i \neq j \rightarrow a\langle i < v \rangle[j] = a[j]\]
The array property fragment can express equality between arrays

This is called **extensionality**

For a quantifier-free formula that asserts \( a = b \) for arrays \( a, b \)
we can write an equisatisfiable formula in the fragment

For any instance of \( a = b \), replace it with:

\[
\forall i. a[i] = b[i]
\]
Observe: Semantically, \( \forall i.F[i] \) is equivalent to
\[
\bigwedge_{\beta} \beta(F)
\]
Where \( \beta \) ranges over all substitutions of \( i \)

Basic idea: Replace \( \forall i.F[i] \) with a finite conjunction \( F[t_1] \land \cdots \land F[t_n] \)

\( t_1, \ldots, t_n \) are called the index terms

Need to find \( t_1, \ldots, t_n \) sufficient to decide satisfiability

For the array property fragment, we can find \( t_1, \ldots, t_n \) by looking at \( F \)
Example

Suppose we want to know if $F$ is satisfiable:

$$F : \forall j. a \langle i < v \rangle [j] = a[j] \land a[i] \neq v$$

Which index terms do we need to consider?

In this case, just $i$:

$$F' : \left( \land_{j \in \{i\}} a \langle i \rightarrow v \rangle [j] = a[j] \right) \land a[i] \neq v$$

This simplifies to:

$$a \langle i < v \rangle [i] = a[i] \land a[i] \neq v$$

Or further:

$$v = a[i] \land a[i] \neq v$$

So, the formula is not satisfiable
Given a formula $F$ in the array property fragment:

1. $F_1$: Put $F$ in negation normal form
2. $F_2$: Remove all write terms from $F_1$
3. $F_3$: Remove existential quantifiers from $F_2$
4. Select a sufficient set of index terms $\mathcal{I}$
5. $F_4$: Transform $\forall$ into conjunction using $\mathcal{I}$
6. Decide the $T_A$-satisfiability of $F_5$
Step 1: Put $F$ in NNF

Apply equivalences to convert to NNF:

$$
\neg \neg F \iff F \\
\neg \top \iff \bot \\
\neg \bot \iff \top \\
\neg (F_1 \land F_2) \iff \neg F_1 \lor \neg F_2 \\
\neg (F_1 \lor F_2) \iff \neg F_1 \land \neg F_2 \\
F_1 \rightarrow F_2 \iff \neg F_1 \lor F_2 \\
\neg \forall x. F[x] \iff \exists x. \neg F[x] \\
\neg \exists x. F[x] \iff \forall x. \neg F[x]
$$

\[\begin{align*}
\langle atom \rangle &::= \top | \bot | P, Q, \ldots \\
\langle literal \rangle &::= \langle atom \rangle | \neg \langle atom \rangle \\
\langle formula \rangle &::= \langle literal \rangle \\
&\quad | \langle formula \rangle \land \langle formula \rangle \\
&\quad | \langle formula \rangle \lor \langle formula \rangle \\
&\quad | \forall x. \langle formula \rangle \\
&\quad | \exists x. \langle formula \rangle
\end{align*}\]
Convert the following to NNF:

\[ \forall x. (\exists y.p(x, y) \land p(x, z)) \rightarrow \exists w.p(x, w) \]

1. \[ \forall x. \neg (\exists y.p(x, y) \land p(x, z)) \lor \exists w.p(x, w) \]
2. \[ \forall x. (\forall y. \neg (p(x, y) \land p(x, z))) \lor \exists w.p(x, w) \]
3. \[ \forall x. (\forall y. \neg p(x, y) \lor \neg p(x, z)) \lor \exists w.p(x, w) \]
Example

Is the following in NNF?

\[ a⟨ℓ □ v⟩[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq ℓ \rightarrow a[i] = b[i]) \]

Yes

Moving on...
Step 2: Remove all write terms

\[
\frac{F[\text{a}(i \leftarrow v)]}{F[\text{a}'] \land \text{a}'[i] = v \land (\forall j. j \neq i \rightarrow a[j] = a'[j])}
\]

For fresh \( a' \)

For each occurrence of a write term \( \text{a}(i \leftarrow v) \) in \( F \):

1. Replace it with \( a' \)
2. Conjoin \( a'[i] = v \) to encode the write’s update
3. Conjoin \( \forall j. j \neq i \rightarrow a[j] = a'[j] \) to preserve other indices
Example

Remove the writes from this formula

\[ a(\ell \prec v)[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq \ell \rightarrow a[i] = b[i]) \]

There is one write term, \( a(\ell \prec v) \)

Replace it with \( a' \), and constrain its properties

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq \ell \rightarrow a[i] = b[i]) \]
\[ \land a'[\ell] = v \land (\forall j. j \neq \ell \rightarrow a'[j] = a[j]) \]
Step 3: Remove existential quantifiers

\[
\frac{F[\exists i. G[i]]}{F[G[j]]} \quad \text{For fresh } j
\]

For each occurrence of \(\exists i. G[i]\) in \(F\), instantiate \(i\) with a free variable.

When deciding satisfiability, free variables are implicitly existential.

But the array property fragment doesn’t allow explicit existentials.

Do we need this step?

Existential quantifiers can arise when we convert to NNF.
Step 5: Select the index terms

\[ I = \bigcup \left\{ \lambda \right\} \cup \{ t \mid t \in F \text{ and } t \text{ is not } \forall \text{-quantified} \} \cup \{ t \mid t \text{ is an } \text{evar} \text{ in an index guard of } F \} \]

In this step, we select symbolic array indices that could be relevant

This set contains:

1. All unquantified terms that index a read, e.g., \( a[t] \)
2. All unquantified terms compared to a universally-quantified variable in an index guard, e.g., \( \ell \) in \( \ell \neq i \)
3. A fresh constant \( \lambda \) that represents index positions not explicitly in \( I \)
Example

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq \ell \rightarrow a[i] = b[i]) \]
\[ \land a'[\ell] = v \land (\forall j. j \neq \ell \rightarrow a'[j] = a[j]) \]

Recalling,

\[ \mathcal{I} = \bigcup \{ \lambda \} \bigcup \{ t \mid [t] \in F \text{ and } t \text{ is not } \forall \text{-quantified} \} \bigcup \{ t \mid t \text{ is an evar in an index guard of } F \} \]

What is \( \mathcal{I} \) in this example?

\[ \mathcal{I} = \{ \lambda, k, \ell \} \]
Exhaustively apply:

\[
\frac{H[\forall i. F[i] \rightarrow G[i]]}{\quad H \left[ \bigwedge_{i \in \mathcal{I}} (F[i] \rightarrow G[i]) \right]}
\]

This is the crux of the algorithm

For each occurrence of \( \forall i. F[i] \rightarrow G[i] \) in \( H \),

1. Replace with the conjunction \( \bigwedge_{i \in \mathcal{I}} (F[i] \rightarrow G[i]) \)
2. In each conjunction, the quantified variable ranges over all terms in \( \mathcal{I} \)

After eliminating \( \forall \), conjoin the following term:

\[
\bigwedge_{i \in \mathcal{I} \setminus \{\lambda\}} \lambda \neq i
\]
Example

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land (\forall i. i \neq \ell \rightarrow a[i] = b[i]) \]
\[ \land a'[\ell] = v \land (\forall j. j \neq \ell \rightarrow a'[j] = a[j]) \]

We found that \( \mathcal{I} = \{\lambda, k, \ell\} \)

What is the equivalent formula free of \( \forall \)?

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land \bigwedge_{i \in \mathcal{I}} (i \neq \ell \rightarrow a[i] = b[i]) \]
\[ \land a'[\ell] = v \land \bigwedge_{j \in \mathcal{I}} (j \neq \ell \rightarrow a'[j] = a[j]) \land \lambda \neq k \land \lambda \neq \ell \]
Example cont’d.

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land \bigwedge_{i \in \mathcal{I}} (i \neq \ell \rightarrow a[i] = b[i]) \]
\[ \land a'[\ell] = v \land \bigwedge_{j \in \mathcal{I}} (j \neq \ell \rightarrow a'[j] = a[j]) \land \lambda \neq k \land \lambda \neq \ell \]

Expanding,

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land a'[\ell] = v \land \]
\[ (\lambda \neq \ell \rightarrow a[\lambda] = b[\lambda]) \land (k \neq \ell \rightarrow a[k] = b[k]) \land (\ell \neq \ell \rightarrow a[\ell] = b[\ell]) \land \]
\[ (\lambda \neq \ell \rightarrow a'[\lambda] = a[\lambda]) \land (k \neq \ell \rightarrow a'[k] = a[k]) \land (\ell \neq \ell \rightarrow a'[\ell] = a[\ell]) \]
\[ \lambda \neq k \land \lambda \neq \ell \]

Simplifying,

\[ a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land a'[\ell] = v \land \lambda \neq k \land \lambda \neq \ell \land \]
\[ a[\lambda] = b[\lambda] \land (k \neq \ell \rightarrow a[k] = b[k]) \land a'[\lambda] = a[\lambda] \land (k \neq \ell \rightarrow a'[k] = a[k]) \]
Step 6: Decide the resulting $T_A$-formula

We left off with:

$$a'[k] = b[k] \land b[k] \neq v \land a[k] = v \land a'[\ell] = v \land \lambda \neq k \land \lambda \neq \ell \land$$

$$a[\lambda] = b[\lambda] \land (k \neq \ell \rightarrow a[k] = b[k]) \land a'[\lambda] = a[\lambda] \land (k \neq \ell \rightarrow a'[k] = a[k])$$

There are two cases two consider:

1. $k = \ell$: But $a'[\ell] = v$ and $a'[k] = b[k]$ imply that $b[k] = v$, but this contradicts $b[k] \neq v$.

2. $k \neq \ell$: But $a[k] = v$ and $a[k] = b[k]$ imply that $b[k] = v$ again.

So, $F$ is unsatisfiable in the array property fragment.
Example

Is the following satisfiable?
\[(\forall i. i \neq j \rightarrow a[i] = b[i]) \land (\forall i. i \neq k \rightarrow a[i] \neq b[i])\]

What is the index set?
\[\mathcal{I} = \{\lambda, j, k\}\]

What is the $\forall$-eliminated formula?
\[(\lambda \neq j \rightarrow a[\lambda] = b[\lambda]) \land (j \neq j \rightarrow a[j] = b[j]) \land (k \neq j \rightarrow a[k] = b[k]) \land (\lambda \neq k \rightarrow a[\lambda] \neq b[\lambda]) \land (j \neq k \rightarrow a[j] \neq b[j]) \land (k \neq k \rightarrow a[k] \neq b[k]) \land \lambda \neq j \land \lambda \neq k\]

$\lambda \neq j, \lambda \neq k, \lambda \neq j \rightarrow a[\lambda] = b[\lambda], \lambda \neq k \rightarrow a[\lambda] \neq b[\lambda]$ contradict each other.
Example

$$(\forall i. i \neq j \rightarrow a[i] = b[i]) \land (\forall i. i \neq k \rightarrow a[i] \neq b[i])$$

What if we didn’t use \(\lambda\) to model the “other” indices?

We’d have the index set: \(\mathcal{I} = \{j, k\}\)

And the \(\forall\)-eliminated formula:

$$(j \neq j \rightarrow a[j] = b[j]) \land (k \neq j \rightarrow a[k] = b[k]) \land$$

$$(j \neq k \rightarrow a[j] \neq b[j]) \land (k \neq k \rightarrow a[k] \neq b[k])$$

This simplifies to:

$$(j \neq k \rightarrow a[k] = b[k]) \land a[j] \neq b[j]$$

Unlike before, this formula is satisfiable
The technique we just looked at works by *instantiating* quantifiers:

\[ \forall x. F[x] \text{ is equivalent to } \bigwedge_{\beta} \beta(F[x]) \]

where \( \beta \) ranges over all substitutions of \( x \)

We considered fragments where one can always find a sufficient **finite** set of symbolic substitutions

Modern solvers use instantiation in general cases as well

**But**, this approach is not complete (hence, *heuristic*)

This approach is sometimes called “E-matching”
Suppose we have the following formula:

\[ z \geq y \land p(z, y) \land (\forall x. p(x, y) \rightarrow x < y) \]

We can unify \( p(x, y) \) with \( p(z, y) \), which binds \( x \mapsto z \)

This binding allows us to instantiate the quantifier:

\[ z \geq y \land p(z, y) \land (\forall x. p(x, y) \rightarrow x < y) \land (p(z, y) \rightarrow z < y) \]

\[ \Leftrightarrow z \geq y \land p(z, y) \land (\forall x. p(x, y) \rightarrow x > y) \land z < y \]

In this case, it was easy to find terms in the formula to unify

Generally, these terms might not be readily available
Triggers: “Hints” for instantiation

Z3, the solver used by Dafny, relies on this techniques

Dafny provides *trigger* patterns to aid instantiation

Dafny walks the AST rooted at a quantifier:

1. Finds terms that can act as triggers to match on: user-defined functions that mention all quantified variables
2. Identifies “killer” terms that prevent parent nodes from acting as triggers: calls to interpreted functions

For example:

Formula: \( \forall x. P(x) \land (Q(x) \rightarrow P(x + 1)) \)

Expressions: \( x, P(x), Q(x), 1, x + 1, P(x + 1), Q(x) \rightarrow P(x + 1) \)

Triggers: \( P(x), Q(x) \)

Killers: \( x + 1, P(x + 1), Q(x) \rightarrow P(x + 1) \)

(Thanks to Clément Pit-Claudel for the example)
Triggers in Dafny

*/\ No terms found to trigger on.

This means that Dafny can’t find any matches

Sometimes, it may just fail to prove what you think it should

You can help it along with trigger annotations

```dafny
predicate P(x:int) { true }

lemma MoreThanOneInt(x: int)
  ensures exists z: int { :trigger P(z) } :: z != x
  {
    assert P(x+1);
  }
```

(Thanks to Tim Wood for the example)
Triggers in Dafny

For more on triggers and Dafny, see:

*Trigger Selection Strategies to Stabilize Program Verifiers,*
K.R.M. Leino and Clément Pit-Claudel

For more on triggers in general, see:

*Programming with Triggers,*
Michal Moskal
For more on today’s material, see Chapter 9 of Bradley & Manna

Next time, we’ll talk about

- Satisfiability Modulo Theories (SMT)
- Induction