universität innsbruck

SS 2024 lecture 13

Constraint Solving

René Thiemann and Fabian Mitterwallner

based on a previous course by Aart Middeldorp

Outline

Outline

Arrays

- when reasoning on arrays, there are two problems

  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Moving Array Elements

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Moving Array Elements

```
int a[N];        // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problems

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Moving Array Elements

```
int a[N];        // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problems
  1. $i < N \to 0 \leq i < N \land 0 \leq i + 1 < N$ (LIA formula)

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Moving Array Elements

```
int a[N];        // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problems
  1. $i < N \to 0 \leq i < N \wedge 0 \leq i + 1 < N$ (LIA formula)
  2. $\forall i.\ 0 < i < N \to a'[i-1] = a[i]$ (array formula)

     where a refers to original array, and a' to array after execution

Arrays

- when reasoning on arrays, there are two problems

  1. are the array accesses within bounds? (this section)
  2. does the array store the intended values? (upcoming sections)

Moving Array Elements

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problems

  1. $i < N \to 0 \leq i < N \land 0 \leq i+1 < N$ (LIA formula)
  2. $\forall i.\ 0 < i < N \to a'[i-1] = a[i]$ (array formula)

     where a refers to original array, and a' to array after execution

Consequence

Checking array bounds does not need special logic about arrays; integer arithmetic suffices

Example (Checking Array-Bounds)

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \rightarrow 0 \leq i < N \wedge 0 \leq i + 1 < N$ is not valid

Example (Checking Array-Bounds)

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \to 0 \leq i < N \wedge 0 \leq i + 1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant

Example (Checking Array-Bounds)

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \to 0 \leq i < N \wedge 0 \leq i + 1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant
  - adding invariant (such as $i \geq 0$) is crucial for proving lower bounds in this example

Example (Checking Array-Bounds)

```
int a[N];        // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \rightarrow 0 \leq i < N \land 0 \leq i + 1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant
  - adding invariant (such as $i \geq 0$) is crucial for proving lower bounds in this example
  - invariant can be used as additional assumption, i.e., formula above becomes $i < n \land i \geq 0 \rightarrow 0 \leq i < N \land 0 \leq i + 1 < N$

Example (Checking Array-Bounds)

```
int a[N];        // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \to 0 \leq i < N \wedge 0 \leq i+1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant
  - adding invariant (such as $i \geq 0$) is crucial for proving lower bounds in this example
  - invariant can be used as additional assumption, i.e., formula above becomes
    $i < n \wedge i \geq 0 \to 0 \leq i < N \wedge 0 \leq i+1 < N$
  - loop invariant itself has to be proven
    - when entering the loop: $i = 0 \to i \geq 0$
    - after each loop iteration: $i < N \wedge i \geq 0 \to i' = i+1 \to i' \geq 0$

Example (Checking Array-Bounds)

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \rightarrow 0 \leq i < N \wedge 0 \leq i + 1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant
  - adding invariant (such as $i \geq 0$) is crucial for proving lower bounds in this example
  - invariant can be used as additional assumption, i.e., formula above becomes $i < n \wedge i \geq 0 \rightarrow 0 \leq i < N \wedge 0 \leq i + 1 < N$
  - loop invariant itself has to be proven
    - when entering the loop: $i = 0 \rightarrow i \geq 0$
    - after each loop iteration: $i < N \wedge i \geq 0 \rightarrow i' = i + 1 \rightarrow i' \geq 0$
- second problem: even with loop invariant, formula is not valid
  - violating assignment shows real bug in program, e.g., `N = 5, i = 4`

Example (Checking Array-Bounds)

```
int a[N];       // an array with entries a[0], ..., a[N-1]
int i = 0;
while (i < N) {  a[i] = a[i+1]; i = i+1; }
```

- problem: formula $i < N \to 0 \leq i < N \wedge 0 \leq i + 1 < N$ is not valid
- first problem: spurious counter-example (`i = -3, N = 7`) $\Longrightarrow$ add loop invariant
  - adding invariant (such as $i \geq 0$) is crucial for proving lower bounds in this example
  - invariant can be used as additional assumption, i.e., formula above becomes $i < n \wedge i \geq 0 \to 0 \leq i < N \wedge 0 \leq i + 1 < N$
  - loop invariant itself has to be proven
    - when entering the loop: $i = 0 \to i \geq 0$
    - after each loop iteration: $i < N \wedge i \geq 0 \to i' = i + 1 \to i' \geq 0$
- second problem: even with loop invariant, formula is not valid
  - violating assignment shows real bug in program, e.g., `N = 5, i = 4`
    - correct `while (i < N)` to `while (i + 1 < N)` in program

Outline

Arrays

- when reasoning on arrays, there are two problems

  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)

- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Arrays

- when reasoning on arrays, there are two problems

  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)

- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, ...)

Arrays

- when reasoning on arrays, there are two problems

  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)

- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, ...)
- array type T_A is just the type $T_I \to T_E$, i.e., maps from index type to element type

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, ...)
- array type T_A is just the type $T_I \to T_E$, i.e., maps from index type to element type
- new primitives in logic (in addition to what is available in index theory and element theory)

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, $\ldots$)
- array type T_A is just the type $T_I \to T_E$, i.e., maps from index type to element type
- new primitives in logic (in addition to what is available in index theory and element theory)
  - array write (array update): $a\{i := e\}$ modified array a where e is written at index i

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, ...)
- array type T_A is just the type $T_I \to T_E$, i.e., maps from index type to element type
- new primitives in logic (in addition to what is available in index theory and element theory)
  - array write (array update): $a\{i := e\}$ modified array a where e is written at index i
  - array read (array index): $a[i]$ read array a at index i

Arrays

- when reasoning on arrays, there are two problems
  1. are the array accesses within bounds? (previous section, now assumed)
  2. does the array store the intended values? (this section)
- for the second problem, we actually need a logic that permits us to describe properties of arrays, in particular basic operations on arrays

Array Logic

- array logic is parametrised by
  - index theory with index type T_I (here: always $\mathbb{Z}$)
  - element theory with element type T_E: content of arrays (here: $\mathbb{Z}$, $\mathbb{B}$, $\ldots$)
- array type T_A is just the type $T_I \to T_E$, i.e., maps from index type to element type
- new primitives in logic (in addition to what is available in index theory and element theory)
  - array write (array update): $a\{i := e\}$ modified array a where e is written at index i
  - array read (array index): $a[i]$ read array a at index i
  - array equality: $a = a'$ compare two arrays

Example (Setting up Verification Conditions)

- program for initializing an array with "true" ($\top$ in mathematical notation)

```
bool a[N];
int i = 0;
while (i < N) { a[i] = true ; i = i+1; }
```

Example (Setting up Verification Conditions)

- program for initializing an array with "true" ($\top$ in mathematical notation)

```
bool a[N];
int i = 0;
while (i < N) { a[i] = true ; i = i+1; }
```

- verification via invariant in this example requires array logic ($T_I = \mathbb{Z}$, $T_E = \mathbb{B}$)

$$\underbrace{(\forall x \in \mathbb{Z}. 0 \leq x < i \to a[x])}_{\text{precondition = invariant}} \wedge \underbrace{a' = a\{i := \top\} \wedge i' = i + 1}_{\text{loop iteration}} \to \underbrace{(\forall x \in \mathbb{Z}. 0 \leq x < i' \to a'[x])}_{\text{postcondition = invariant for }'\text{-variables}}$$

Example (Setting up Verification Conditions)

- program for initializing an array with "true" ($\top$ in mathematical notation)

```
bool a[N];
int i = 0;
while (i < N) { a[i] = true ; i = i+1; }
```

- verification via invariant in this example requires array logic ($T_I = \mathbb{Z}$, $T_E = \mathbb{B}$)

$$\underbrace{(\forall x \in \mathbb{Z}.0 \leq x < i \rightarrow a[x])}_{\text{precondition = invariant}} \wedge \underbrace{a' = a\{i := \top\} \wedge i' = i + 1}_{\text{loop iteration}} \rightarrow \underbrace{(\forall x \in \mathbb{Z}.0 \leq x < i' \rightarrow a'[x])}_{\text{postcondition = invariant for }'\text{-variables}}$$

Observations

- reasoning about array logic formulas requires theories about indices and elements

Example (Setting up Verification Conditions)

- program for initializing an array with "true" ($\top$ in mathematical notation)

```
bool a[N];
int i = 0;
while (i < N) { a[i] = true ; i = i+1; }
```

- verification via invariant in this example requires array logic ($T_I = \mathbb{Z}$, $T_E = \mathbb{B}$)

$$\underbrace{(\forall x \in \mathbb{Z}.0 \leq x < i \to a[x])}_{\text{precondition = invariant}} \wedge \underbrace{a' = a\{i := \top\} \wedge i' = i + 1}_{\text{loop iteration}} \to \underbrace{(\forall x \in \mathbb{Z}.0 \leq x < i' \to a'[x])}_{\text{postcondition = invariant for }'\text{-variables}}$$

Observations

- reasoning about array logic formulas requires theories about indices and elements
  - index theory usually requires quantifiers (each/some array element satisfies property)

Example (Setting up Verification Conditions)

- program for initializing an array with "true" ($\top$ in mathematical notation)

```
bool a[N];
int i = 0;
while (i < N) { a[i] = true ; i = i+1; }
```

- verification via invariant in this example requires array logic ($T_I = \mathbb{Z}$, $T_E = \mathbb{B}$)

$$\underbrace{(\forall x \in \mathbb{Z}. 0 \leq x < i \rightarrow a[x])}_{\text{precondition = invariant}} \wedge \underbrace{a' = a\{i := \top\} \wedge i' = i + 1}_{\text{loop iteration}} \rightarrow \underbrace{(\forall x \in \mathbb{Z}. 0 \leq x < i' \rightarrow a'[x])}_{\text{postcondition = invariant for }'\text{-variables}}$$

Observations

- reasoning about array logic formulas requires theories about indices and elements
  - index theory usually requires quantifiers (each/some array element satisfies property)
  - suitable choice: Presburger arithmetic (linear arithmetic over $\mathbb{Z}$ with quantifiers)

Semantics of Array Logic (meaning of array-index, -update, -equality)

- array congruence: if arrays are equal and indices are equal, then identical elements are obtained when reading from an array

$$\forall a, b \in T_A, i, j \in T_I.\ a = b \rightarrow i = j \rightarrow a[i] = b[j] \tag{1}$$

Semantics of Array Logic (meaning of array-index, -update, -equality)

- array congruence: if arrays are equal and indices are equal, then identical elements are obtained when reading from an array

$$\forall a, b \in T_A, i, j \in T_I.\ a = b \rightarrow i = j \rightarrow a[i] = b[j] \tag{1}$$

- array-updates: read-over-write axiom

$$\forall a \in T_A, e \in T_E, i, j \in T_I.\ a\{i := e\}[j] = \begin{cases} e, & \text{if } i = j \\ a[j], & \text{otherwise} \end{cases} \tag{2}$$

Semantics of Array Logic (meaning of array-index, -update, -equality)

- array congruence: if arrays are equal and indices are equal, then identical elements are obtained when reading from an array

$$\forall a, b \in T_A, i, j \in T_I.\ a = b \to i = j \to a[i] = b[j] \tag{1}$$

- array-updates: read-over-write axiom

$$\forall a \in T_A, e \in T_E, i, j \in T_I.\ a\{i := e\}\,[j] = \begin{cases} e, & \text{if } i = j \\ a[j], & \text{otherwise} \end{cases} \tag{2}$$

- optional extensionality rule: two arrays are equal if they store the same elements

$$\forall a, b \in T_A.\ (\forall i \in T_I.\ a[i] = b[i]) \to a = b \tag{3}$$

Eliminating the Array Terms

- aim: translate formula in array logic to formula over
  - index theory,
  - element theory, and
  - uninterpreted functions

  in order to use decision procedure for this combination for array logic formulas

Eliminating the Array Terms

- aim: translate formula in array logic to formula over
  - index theory,
  - element theory, and
  - uninterpreted functions

  in order to use decision procedure for this combination for array logic formulas
- main idea

Eliminating the Array Terms

- aim: translate formula in array logic to formula over
  - index theory,
  - element theory, and
  - uninterpreted functions

  in order to use decision procedure for this combination for array logic formulas
- main idea
  - arrays behave like uninterpreted functions: according to (1), reading an array at the same index yields same elements; function invocations on same inputs return same result

Eliminating the Array Terms

- aim: translate formula in array logic to formula over
  - index theory,
  - element theory, and
  - uninterpreted functions

  in order to use decision procedure for this combination for array logic formulas
- main idea
  - arrays behave like uninterpreted functions: according to (1), reading an array at the same index yields same elements; function invocations on same inputs return same result
  - translation
    - for each array a introduce corresponding unary uninterpreted function A
    - array read access $a[i]$ is translated to function application $A(i)$

Example (Eliminating Array Terms)

- consider array logic formula with element type being characters

$$i = j \rightarrow a[j] = \text{'c'} \rightarrow a[i] = \text{'c'}$$

Example (Eliminating Array Terms)

- consider array logic formula with element type being characters

$$i = j \rightarrow a[j] = \text{'}c\text{'} \rightarrow a[i] = \text{'}c\text{'}$$

- elimination results in formula

$$i = j \rightarrow A(j) = \text{'}c\text{'} \rightarrow A(i) = \text{'}c\text{'}$$

Example (Eliminating Array Terms)

- consider array logic formula with element type being characters

$$i = j \rightarrow a[j] = \text{'}c\text{'} \rightarrow a[i] = \text{'}c\text{'}$$

- elimination results in formula

$$i = j \rightarrow A(j) = \text{'}c\text{'} \rightarrow A(i) = \text{'}c\text{'}$$

- validity of formula can be shown by decision procedure for equality and uninterpreted functions (EUF)

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\ j \neq i \rightarrow b[j] = a[j]$

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\ j \neq i \rightarrow b[j] = a[j]$
- write rule is an equivalence preserving transformation

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\, j \neq i \rightarrow b[j] = a[j]$
- write rule is an equivalence preserving transformation

Example (requiring first constraint)

- formula $a\{i := e\}[i] + 2 \geq e$ is translated into

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\ j \neq i \rightarrow b[j] = a[j]$
- write rule is an equivalence preserving transformation

Example (requiring first constraint)

- formula $a\{i := e\}[i] + 2 \geq e$ is translated into

$$b[i] = e \wedge (\forall j.\ j \neq i \rightarrow b[j] = a[j]) \rightarrow b[i] + 2 \geq e$$

Eliminating the Array Terms – Array Updates

- aim: translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\, j \neq i \rightarrow b[j] = a[j]$
- write rule is an equivalence preserving transformation

Example (requiring first constraint)

- formula $a\{i := e\}[i] + 2 \geq e$ is translated into

$$b[i] = e \wedge (\forall j.\, j \neq i \rightarrow b[j] = a[j]) \rightarrow b[i] + 2 \geq e$$

  whose validity is easily proven:
  apply equality $b[i] = e$ and prove resulting LIA constraint $e + 2 \geq e$

Eliminating the Array Terms – Array Updates, continued

- translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\, j \neq i \rightarrow b[j] = a[j]$

Example (requiring second constraint)

Eliminating the Array Terms – Array Updates, continued

- translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\, j \neq i \rightarrow b[j] = a[j]$

Example (requiring second constraint)

- formula $a[0] = 5 \rightarrow a\{7 := x + 1\}[0] = 5$ is translated into

$$b[7] = x + 1 \wedge (\forall j.\, j \neq 7 \rightarrow b[j] = a[j]) \wedge a[0] = 5 \rightarrow b[0] = 5$$

Eliminating the Array Terms – Array Updates, continued

- translate $a\{i := e\}$ via write rule:
  - replace an occurrence of $a\{i := e\}$ by a fresh array variable b
  - add two constraints that describe relationship between a and b by using (2)
    - $b[i] = e$
    - $\forall j.\, j \neq i \rightarrow b[j] = a[j]$

Example (requiring second constraint)

- formula $a[0] = 5 \rightarrow a\{7 := x + 1\}\,[0] = 5$ is translated into

$$b[7] = x + 1 \wedge (\forall j.\, j \neq 7 \rightarrow b[j] = a[j]) \wedge a[0] = 5 \rightarrow b[0] = 5$$

  whose validity can easily be proven in EUF + LIA after its translation

$$B(7) = x + 1 \wedge (\forall j.\, j \neq 7 \rightarrow B(j) = A(j)) \wedge A(0) = 5 \rightarrow B(0) = 5$$

Elimination of Array Terms – A Problem

- array terms can easily be eliminated; resulting formulas are combination of
  - index theory + quantification
  - element theory
  - uninterpreted functions

Elimination of Array Terms – A Problem

- array terms can easily be eliminated; resulting formulas are combination of
  - index theory + quantification
  - element theory
  - uninterpreted functions
- problem: even if
  - index theory + quantification
  - element theory

  is decidable, the combination with uninterpreted functions is not necessarily decidable

Elimination of Array Terms – A Problem

- array terms can easily be eliminated; resulting formulas are combination of
  - index theory + quantification
  - element theory
  - uninterpreted functions
- problem: even if
  - index theory + quantification
  - element theory

  is decidable, the combination with uninterpreted functions is not necessarily decidable
- example
  - choose index theory = element theory = Presburger arithmetic (decidable)

Elimination of Array Terms – A Problem

- array terms can easily be eliminated; resulting formulas are combination of
  - index theory + quantification
  - element theory
  - uninterpreted functions
- problem: even if
  - index theory + quantification
  - element theory

  is decidable, the combination with uninterpreted functions is not necessarily decidable
- example
  - choose index theory = element theory = Presburger arithmetic (decidable)
  - when adding uninterpreted functions, this becomes undecidable

Elimination of Array Terms – A Problem

- array terms can easily be eliminated; resulting formulas are combination of
  - index theory + quantification
  - element theory
  - uninterpreted functions
- problem: even if
  - index theory + quantification
  - element theory

  is decidable, the combination with uninterpreted functions is not necessarily decidable
- example
  - choose index theory = element theory = Presburger arithmetic (decidable)
  - when adding uninterpreted functions, this becomes undecidable
- potential solution: do not allow all array logic formulas, but a decidable fragment

Outline

Array Properties

- restricted class of array logic formulas; decidable fragment

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

$$\forall\, i_1, \dots, i_k \in T_I.\ \varphi_I(i_1, \dots, i_k) \rightarrow \varphi_V(i_1, \dots, i_k)$$

where

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

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

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

$$\forall\, i_1, \dots, i_k \in T_I.\ \varphi_I(i_1, \dots, i_k) \to \varphi_V(i_1, \dots, i_k)$$

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free
  - index guard is formula consisting of Boolean disjunction, conjunction, and comparison of iterms via $\leq$ or $=$

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

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

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free
  - index guard is formula consisting of Boolean disjunction, conjunction, and comparison of iterms via $\leq$ or $=$
  - iterm is either $i_1, \ldots, i_k$ or a linear integer expression e with $\mathit{vars}(e)$ disjoint from $i_1, \ldots, i_k$

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

$$\forall\, i_1, \dots, i_k \in T_I.\ \varphi_I(i_1, \dots, i_k) \to \varphi_V(i_1, \dots, i_k)$$

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free
  - index guard is formula consisting of Boolean disjunction, conjunction, and comparison of iterms via $\leq$ or $=$
  - iterm is either $i_1, \dots, i_k$ or a linear integer expression e with $vars(e)$ disjoint from $i_1, \dots, i_k$
  - $i_i, \dots, i_k$ may only be used in array read accesses of form $a[i_j]$ within value constraint

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

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

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free
  - index guard is formula consisting of Boolean disjunction, conjunction, and comparison of iterms via $\leq$ or $=$
  - iterm is either $i_1, \ldots, i_k$ or a linear integer expression e with $vars(e)$ disjoint from $i_1, \ldots, i_k$
  - $i_i, \ldots, i_k$ may only be used in array read accesses of form $a[i_j]$ within value constraint
- fragment restricts formulas to Boolean combination of array properties

Array Properties

- restricted class of array logic formulas; decidable fragment
- formula is array property if it is of the form

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

  where

  - φ_I is called index guard, φ_V is value constraint, both are quantifier-free
  - index guard is formula consisting of Boolean disjunction, conjunction, and comparison of iterms via $\leq$ or $=$
  - iterm is either $i_1, \ldots, i_k$ or a linear integer expression e with $vars(e)$ disjoint from $i_1, \ldots, i_k$
  - $i_i, \ldots, i_k$ may only be used in array read accesses of form $a[i_j]$ within value constraint
- fragment restricts formulas to Boolean combination of array properties
- free variables are implicitly existentially quantified

Example

Consider negated (simplified) verification condition from before; aim: show unsatisfiability

$$\underbrace{(\forall x \in \mathbb{Z}.\ x < i \to a[x])}_{\text{precondition}} \wedge \underbrace{a' = a\{i := \top\}}_{\text{loop iteration}} \wedge \underbrace{\neg(\forall x \in \mathbb{Z}.\ x < i + 1 \to a'[x])}_{\text{negated postcondition}}$$

Example

Consider negated (simplified) verification condition from before; aim: show unsatisfiability

$$\underbrace{(\forall x \in \mathbb{Z}.\, x < i \to a[x])}_{\text{precondition}} \wedge \underbrace{a' = a\{i := \top\}}_{\text{loop iteration}} \wedge \underbrace{\neg(\forall x \in \mathbb{Z}.\, x < i + 1 \to a'[x])}_{\text{negated postcondition}}$$

- loop iteration is already array property

Example

Consider negated (simplified) verification condition from before; aim: show unsatisfiability

$$\underbrace{(\forall x \in \mathbb{Z}.\ x < i \rightarrow a[x])}_{\text{precondition}} \wedge \underbrace{a' = a\{i := \top\}}_{\text{loop iteration}} \wedge \underbrace{\neg(\forall x \in \mathbb{Z}.\ x < i + 1 \rightarrow a'[x])}_{\text{negated postcondition}}$$

- loop iteration is already array property
- precondition and postcondition are nearly array properties, just need to eliminate $<$

Example

Consider negated (simplified) verification condition from before; aim: show unsatisfiability

$$\underbrace{(\forall x \in \mathbb{Z}.\, x < i \to a[x])}_{\text{precondition}} \wedge \underbrace{a' = a\{i := \top\}}_{\text{loop iteration}} \wedge \underbrace{\neg(\forall x \in \mathbb{Z}.\, x < i+1 \to a'[x])}_{\text{negated postcondition}}$$

- loop iteration is already array property
- precondition and postcondition are nearly array properties, just need to eliminate $<$
- resulting formula within fragment

$$(\forall x \in \mathbb{Z}.\, x \leq i-1 \to a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x \in \mathbb{Z}.\, x \leq i \to a'[x])$$

Example

Consider negated (simplified) verification condition from before; aim: show unsatisfiability

$$\underbrace{(\forall x \in \mathbb{Z}.\ x < i \rightarrow a[x])}_{\text{precondition}} \wedge \underbrace{a' = a\{i := \top\}}_{\text{loop iteration}} \wedge \underbrace{\neg(\forall x \in \mathbb{Z}.\ x < i + 1 \rightarrow a'[x])}_{\text{negated postcondition}}$$

- loop iteration is already array property
- precondition and postcondition are nearly array properties, just need to eliminate $<$
- resulting formula within fragment

$$(\forall x \in \mathbb{Z}.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x \in \mathbb{Z}.\ x \leq i \rightarrow a'[x])$$

- note that replacing $x < i$ by $x + 1 \leq i$ does not work, since $x + 1$ is no iterm; reason: x is universally quantified

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
    1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ
  4. replace each universal quantification $\forall i \in T_i.\ P(i)$ within formula φ by finite conjunction $\bigwedge i \in \mathcal{I}(\varphi).\ P(i)$ where $\mathcal{I}(\varphi)$ is set of index terms that i might possibly equal to

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ
  4. replace each universal quantification $\forall i \in T_i.\ P(i)$ within formula φ by finite conjunction $\bigwedge i \in \mathcal{I}(\varphi).\ P(i)$ where $\mathcal{I}(\varphi)$ is set of index terms that i might possibly equal to
     - if $a[e]$ is an array read access in φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ
  4. replace each universal quantification $\forall i \in T_I.\, P(i)$ within formula φ by finite conjunction $\bigwedge i \in \mathcal{I}(\varphi).\, P(i)$ where $\mathcal{I}(\varphi)$ is set of index terms that i might possibly equal to
     - if $a[e]$ is an array read access in φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$
     - if e is an iterm in the index guard of φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ
  4. replace each universal quantification $\forall i \in T_I.\ P(i)$ within formula φ by finite conjunction $\bigwedge i \in \mathcal{I}(\varphi).\ P(i)$ where $\mathcal{I}(\varphi)$ is set of index terms that i might possibly equal to
     - if $a[e]$ is an array read access in φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$
     - if e is an iterm in the index guard of φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$
     - if the previous two rules are not applicable, then define $\mathcal{I}(\varphi) = \{0\}$ to have a non-empty set

Reduction Algorithm for $T_I = LIA$

- translates formula in array logic fragment into equisatisfiable quantifier-free formula over index theory and element theory combined with EUF
- algorithm
  1. convert Boolean formula over array properties to negation normal form (NNF); further convert $\neg\forall$ into $\exists\neg$
  2. replace all array updates via write rule and transform constraints into array properties
  3. remove each existential quantifier by introducing a fresh variable; result is formula φ
  4. replace each universal quantification $\forall i \in T_i.\ P(i)$ within formula φ by finite conjunction $\bigwedge i \in \mathcal{I}(\varphi).\ P(i)$ where $\mathcal{I}(\varphi)$ is set of index terms that i might possibly equal to
     - if $a[e]$ is an array read access in φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$
     - if e is an iterm in the index guard of φ and e is not a quantified variable, then add e to $\mathcal{I}(\varphi)$
     - if the previous two rules are not applicable, then define $\mathcal{I}(\varphi) = \{0\}$ to have a non-empty set
  5. replace array read access operations by uninterpreted functions

Example Reduction Algorithm

- input:

$$(\forall x.\, x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\, x \leq i \rightarrow a'[x])$$

Example Reduction Algorithm

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- conversion to NNF: (push negations inside quantifiers)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

Example Reduction Algorithm

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- conversion to NNF: (push negations inside quantifiers)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

- apply write rule: (eliminate $a' = a\{i := \top\}$, use $a'[i]$ instead of official $a'[i] = \top$)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \neq i \rightarrow a'[j] = a[j]) \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

Example Reduction Algorithm

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- conversion to NNF: (push negations inside quantifiers)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

- apply write rule: (eliminate $a' = a\{i := \top\}$, use $a'[i]$ instead of official $a'[i] = \top$)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \neq i \rightarrow a'[j] = a[j]) \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

- convert constraint to array property: (eliminate $\neq$)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge (\exists x.\ x \leq i \wedge \neg a'[x])$$

Example Reduction Algorithm

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a' = a\{i := \top\} \land \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- conversion to NNF: (push negations inside quantifiers)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a' = a\{i := \top\} \land (\exists x.\ x \leq i \land \neg a'[x])$$

- apply write rule: (eliminate $a' = a\{i := \top\}$, use $a'[i]$ instead of official $a'[i] = \top$)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a'[i] \land (\forall j.\ j \neq i \rightarrow a'[j] = a[j]) \land (\exists x.\ x \leq i \land \neg a'[x])$$

- convert constraint to array property: (eliminate $\neq$)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a'[i] \land (\forall j.\ j \leq i - 1 \lor i + 1 \leq j \rightarrow a'[j] = a[j]) \land (\exists x.\ x \leq i \land \neg a'[x])$$

- remove existential quantifier: (eliminate $\exists x$ by fresh z)

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a'[i] \land (\forall j.\ j \leq i - 1 \lor i + 1 \leq j \rightarrow a'[j] = a[j]) \land z \leq i \land \neg a'[z]$$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \to a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \to a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \to a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \to a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- construct $\mathcal{I}(\varphi)$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \to a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \to a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \to a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \to a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- construct $\mathcal{I}(\varphi)$
  - add i because of $a'[i]$

Example Reduction Algorithm, continued

- input:
$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$
- result of step 3 is formula φ
$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$
- construct $\mathcal{I}(\varphi)$
  - add i because of $a'[i]$
  - add z because of $a'[z]$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- construct $\mathcal{I}(\varphi)$
  - add i because of $a'[i]$
  - add z because of $a'[z]$
  - add $i - 1$ because of $x \leq i - 1$ and $j \leq i - 1$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- construct $\mathcal{I}(\varphi)$
  - add i because of $a'[i]$
  - add z because of $a'[z]$
  - add $i - 1$ because of $x \leq i - 1$ and $j \leq i - 1$
  - add $i + 1$ because of $i + 1 \leq j$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\forall j.\ j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- construct $\mathcal{I}(\varphi) = \{i, z, i - 1, i + 1\}$
  - add i because of $a'[i]$
  - add z because of $a'[z]$
  - add $i - 1$ because of $x \leq i - 1$ and $j \leq i - 1$
  - add $i + 1$ because of $i + 1 \leq j$

Example Reduction Algorithm, continued

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a' = a\{i := \top\} \land \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- result of step 3 is formula φ

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \land a'[i] \land (\forall j.\ j \leq i - 1 \lor i + 1 \leq j \rightarrow a'[j] = a[j]) \land z \leq i \land \neg a'[z]$$

- construct $\mathcal{I}(\varphi) = \{i, z, i - 1, i + 1\}$
  - add i because of $a'[i]$
  - add z because of $a'[z]$
  - add $i - 1$ because of $x \leq i - 1$ and $j \leq i - 1$
  - add $i + 1$ because of $i + 1 \leq j$
- replace universal quantifier:

$$\Big(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]\Big) \land a'[i] \land \Big(\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \lor i + 1 \leq j \rightarrow a'[j] = a[j]\Big) \land z \leq i \land \neg a'[z]$$

Example Reduction Algorithm, completed

- input:

$$(\forall x.\ x \leq i - 1 \to a[x]) \land a' = a\{i := \top\} \land \neg(\forall x.\ x \leq i \to a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i-1, i+1\}$:

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \to a[x]) \land a'[i] \land (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \lor i + 1 \leq j \to a'[j] = a[j]) \land z \leq i \land \neg a'[z]$$

Example Reduction Algorithm, completed

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i-1, i+1\}$:

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- final formula: replace array read access by uninterpreted functions

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow A(x)) \wedge A'(i) \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow A'(j) = A(j)) \wedge z \leq i \wedge \neg A'(z)$$

Example Reduction Algorithm, completed

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i-1, i+1\}$:

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- final formula: replace array read access by uninterpreted functions

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow A(x)) \wedge A'(i) \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow A'(j) = A(j)) \wedge z \leq i \wedge \neg A'(z)$$

- unsatisfiability now decidable: consider cases $z \leq i - 1 \vee z = i \vee z \geq i + 1$ via LIA reasoning

Example Reduction Algorithm, completed

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i - 1, i + 1\}$:

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- final formula: replace array read access by uninterpreted functions

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow A(x)) \wedge A'(i) \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow A'(j) = A(j)) \wedge z \leq i \wedge \neg A'(z)$$

- unsatisfiability now decidable: consider cases $z \leq i - 1 \vee z = i \vee z \geq i + 1$ via LIA reasoning
  - case $z = i$: show unsatisfiability using $A'(i)$ and $\neg A'(z)$ via EUF

Example Reduction Algorithm, completed

- input:

$$(\forall x.\ x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\ x \leq i \rightarrow a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i-1, i+1\}$:

$$\Big(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]\Big) \wedge a'[i] \wedge \Big(\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]\Big) \wedge z \leq i \wedge \neg a'[z]$$

- final formula: replace array read access by uninterpreted functions

$$\Big(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow A(x)\Big) \wedge A'(i) \wedge \Big(\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow A'(j) = A(j)\Big) \wedge z \leq i \wedge \neg A'(z)$$

- unsatisfiability now decidable: consider cases $z \leq i - 1 \vee z = i \vee z \geq i + 1$ via LIA reasoning
  - case $z = i$: show unsatisfiability using $A'(i)$ and $\neg A'(z)$ via EUF
  - case $z \leq i - 1$: since $z \in \mathcal{I}(\varphi)$, obtain $A(z)$, $A'(z) = A(z)$, and $\neg A'(z)$ and use EUF

Example Reduction Algorithm, completed

- input:

$$(\forall x.\, x \leq i - 1 \rightarrow a[x]) \wedge a' = a\{i := \top\} \wedge \neg(\forall x.\, x \leq i \rightarrow a'[x])$$

- formula after quantifier elimination, where $\mathcal{I}(\varphi) = \{i, z, i-1, i+1\}$:

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow a[x]) \wedge a'[i] \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]) \wedge z \leq i \wedge \neg a'[z]$$

- final formula: replace array read access by uninterpreted functions

$$(\bigwedge_{x \in \mathcal{I}(\varphi)} x \leq i - 1 \rightarrow A(x)) \wedge A'(i) \wedge (\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow A'(j) = A(j)) \wedge z \leq i \wedge \neg A'(z)$$

- unsatisfiability now decidable: consider cases $z \leq i - 1 \vee z = i \vee z \geq i + 1$ via LIA reasoning
  - case $z = i$: show unsatisfiability using $A'(i)$ and $\neg A'(z)$ via EUF
  - case $z \leq i - 1$: since $z \in \mathcal{I}(\varphi)$, obtain $A(z)$, $A'(z) = A(z)$, and $\neg A'(z)$ and use EUF
  - case $z \geq i + 1$: show unsatisfiability in combination with $z \leq i$ via LIA

Theorem (Correctness of Reduction Algorithm)

The input formula and the result of the reduction algorithm are equisatisfiable.

Corollary

If satisfiability of quantifier-free $T_{EUF} \cup T_{LIA} \cup T_E$ formulas is decidable, then so is satisfiability of the fragment of array logic for T_E.

A Problem and its Solution

- in the reduction algorithm, the universal part of the write rule

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

  is turned into a finite conjunction

$$\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \to a'[j] = a[j]$$

- problem: this formula often gets (too) large

A Problem and its Solution

- in the reduction algorithm, the universal part of the write rule

$$\forall j.\, j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]$$

  is turned into a finite conjunction

$$\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]$$

- problem: this formula often gets (too) large
- observation: implications are often only required for a few index terms within $\mathcal{I}(\varphi)$ (in previous example, only the index term z was required to prove unsatisfiability)

A Problem and its Solution

- in the reduction algorithm, the universal part of the write rule

$$\forall j.\, j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]$$

  is turned into a finite conjunction

$$\bigwedge_{j \in \mathcal{I}(\varphi)} j \leq i - 1 \vee i + 1 \leq j \rightarrow a'[j] = a[j]$$

- problem: this formula often gets (too) large
- observation: implications are often only required for a few index terms within $\mathcal{I}(\varphi)$ (in previous example, only the index term z was required to prove unsatisfiability)
- solution: use a lazy encoding procedure, that generates instances only on demand, and can be combined with DPLL(T)
- details: see literature, in particular Section 7.4 of Decision Procedures book

Outline

Summary

- checking array bounds is easily encoded via LIA, does not require extension of logic
- array logic provides primitives for array read- and write-accesses
- arrays are easily modeled as uninterpreted functions
- array logic is often undecidable, even for decidable index- and element-theories such as Presburger arithmetic
- array properties define a fragment of array logic;
  the fragment can be translated to quantifier-free formulas by adding EUF
- optimization: lazy encoding creates instances of the write rule on demand

Kröning and Strichmann

- Sections 7.1–7.3
- warning: mistake in example at end of Section 7.3 (over-simplification)
  - problem: $<$ not eliminated
  - result of mistake: smaller set of index terms $\mathcal{I}(\varphi) = \{i, z\}$, but correct set is $\{i, i - 1, z\}$
  - incorrect set does not cause problems in example, but in general elimination of $<$ is essential

Further Reading

Aaron R. Bradley, Zohar Manna, Henny B. Sipma
What's Decidable About Arrays?
Proc. VMCAI 2006, volume 3855 of LNCS, pages 427-–442, 2006

Important Concepts

- array logic
- array property
- checking array bounds via LIA
- invariants
- reduction algorithm
- spurious counterexample
- write rule