

Testing, Debugging, Program Verification

Formal Verification, Part III

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

5 December 2012

Arrays in the While Language

Locations with Array Type

$ArrayLocation ::= Identifier [IntExp]$

Result type is `int` or `boolean` depending on array declaration

Declaration of Arrays in `.key` file

```
\arrays {  
  int[] a;  
  boolean[] b;  
}
```

Arrays in the While Language

Locations with Array Type

$ArrayLocation ::= Identifier [IntExp]$

Result type is `int` or `boolean` depending on array declaration

Declaration of Arrays in `.key` file

```
\arrays {  
  int[] a;  
  boolean[] b;  
}
```

Main Properties of Arrays in While

- ▶ **Value** types
 - ▶ $a[i], b[j]$ different memory locations for all i, j
 - ▶ Array identifier such as a alone is not a location (" $a=b;$ " illegal)
- ▶ **Unbounded**: $a[i]$ valid location for all $i \in \mathbb{Z}$

Representing Arrays in Logic

How to Represent Arrays in the Logic?

An array is a **mapping** from integers to the result type \Rightarrow **non-rigid function** with one integer argument.

Recall: Value of non-rigid function may change. I.e. $a[i]$ may return different values in different states.

Representing Arrays in Logic

How to Represent Arrays in the Logic?

An array is a **mapping** from integers to the result type \Rightarrow **non-rigid function** with one integer argument.

Recall: Value of non-rigid function may change. I.e. $a[i]$ may return different values in different states.

Remembering “Old” Values for Arrays

Introduce a user-defined (rigid) **function** with one integer argument:

```
\functions{  
    int a0(int);  
}
```

Example of this later.

Representing Arrays in Logic

How to Represent Arrays in the Logic?

An array is a **mapping** from integers to the result type \Rightarrow **non-rigid function** with one integer argument.

Recall: Value of non-rigid function may change. I.e. $a[i]$ may return different values in different states.

Remembering “Old” Values for Arrays

Introduce a user-defined (rigid) **function** with one integer argument:

```
\functions{  
    int a0(int);  
}
```

Example of this later.

In Java the Situation is More Complicated

- ▶ Different variables can reference the same array (aliasing)
- ▶ Exceptions are thrown when array index out of bound

Assignments to Array Locations

Assignment Rule is Unchanged!

$$\text{assignment} \frac{\{P\} [\mathcal{U}, x := e] \pi \{Q\}}{\{P\} [\mathcal{U}] x = e; \pi \{Q\}}$$

Works just as well when x is array location

Assignments to Array Locations

Assignment Rule is Unchanged!

$$\text{assignment} \quad \frac{\{P\} [\mathcal{U}, x := e] \pi \{Q\}}{\{P\} [\mathcal{U}] x = e; \pi \{Q\}}$$

Works just as well when x is array location

Update Simplification for Array Locations

What is the result of $[a[i] := 1, a[j] := 2]$?

It depends on whether $i = j$ holds!

$i = j$ Result is $[a[i] := 2]$

$!(i = j)$ Result is $[a[i] := 1 \parallel a[j] := 2]$

KeY-Hoare introduces conditional expressions to **unalias** array indices:

$[a[i] := e](a[j]) \implies \text{\texttt{\textbackslash if}} (i=j) \text{\texttt{\textbackslash then}} (e) \text{\texttt{\textbackslash else}} (a[j])$

Example: Unaliasing of Array Indices

Example (arrayAssign)

```
{ true }  
[]  
a[i] = x;  
a[j] = y;  
{ a[i] = x & a[j] = y }
```

Is this contract valid?

Demo arrayAssign.key

Example: Unaliasing of Array Indices

Example (arrayAssign)

```
{ true }  
[]  
a[i] = x;  
a[j] = y;  
{ a[i] = x & a[j] = y }
```

Is this contract valid?

Demo arrayAssign.key

- ▶ Need to exclude array index aliasing in precondition ($i \neq j$)
- ▶ Open FOL proof goals can give valuable hint

Specifying Contracts with Arrays

Example (binSearch Demo)

```
{ a[l] < x & x < a[r] & "a is sorted" }
```

```
[]
```

```
while (l <= r - 2) {
```

```
    m = (r + l) / 2;
```

```
    if (a[m] < x) {
```

```
        l = m;
```

```
    } else {
```

```
        if (a[m] > x) {
```

```
            r = m;
```

```
        } else {
```

```
            l = m; r = m;
```

```
    } } }
```

```
{ l = r & a[l] = x | l + 1 = r &
```

```
    \forall int i; (i <= l -> a[i] != x) &
```

```
    \forall int i; (i >= r -> a[i] != x) }
```

Intuition: Partial and Total Correctness

Partial Correctness:

- ▶ Program is correct **if it terminates**.
I.e. satisfies contract.
- ▶ But, no requirements on termination.

Total Correctness:

- ▶ Program is correct and **required to terminate**.

Partial and Total Correctness

Another way of thinking about it...

Definition (Partial Correctness)

Program π is **partially correct** wrt P , \mathcal{U} , and Q when $\{P\} [\mathcal{U}] \pi \{Q\}$ is valid Hoare triple with updates.

$\{P\} [\mathcal{U}] \pi \{Q\}$ is valid whenever:

- ▶ P is true in some state s , **and**
- ▶ π terminates if started in \mathcal{U}^s **then**
- ▶ Q is true in the final state $\pi^{\mathcal{U}^s}$.

Partial and Total Correctness

Another way of thinking about it...

Definition (Partial Correctness)

Program π is **partially correct** wrt P , \mathcal{U} , and Q when $\{P\} [\mathcal{U}] \pi \{Q\}$ is valid Hoare triple with updates.

$\{P\} [\mathcal{U}] \pi \{Q\}$ is valid whenever:

- ▶ P is true in some state s , **and**
- ▶ π terminates if started in \mathcal{U}^s **then**
- ▶ Q is true in the final state $\pi^{\mathcal{U}^s}$.

Definition (Total Correctness)

Program π is **totally correct** wrt P , \mathcal{U} , and Q if whenever:

- ▶ P is true in some state s , **then**
- ▶ π terminates if started in \mathcal{U}^s **and**
- ▶ Q is true in the final state $\pi^{\mathcal{U}^s}$.

Partial and Total Correctness

Semantics of Programs: State Transformers

For a program π define **partial function** $\pi^s: \text{State} \rightarrow \text{State}$ as follows:

π^s is the final state of π when started in s , if π terminates, and is undefined otherwise

Partial and Total Correctness (Informally)

Validity of $\{P\} [\mathcal{U}] \pi \{Q\}$ is correctness wrt **partial** function π^s :

\Rightarrow **Partial Correctness**

If we demand in addition that π^s is **total**: \Rightarrow **Total Correctness**

Loop Invariant Rule Is Not Sufficient!

Example (Invariant w/o termination validates any postcondition)

```
{i = 0}
```

```
[]
```

```
while (i >= 0) {}
```

```
{i = 42}
```

Try $i = 0$ as invariant

Loop Invariant Rule Is Not Sufficient!

Example (Invariant w/o termination validates any postcondition)

$\{i = 0\}$

$[]$

while ($i \geq 0$) $\{\}$

$\{i = 42\}$

Try $i = 0$ as invariant

Invariant Rule, Instantiated

$\vdash i = 0 \rightarrow i = 0$

(initially valid) ✓

$\{i = 0 \ \& \ i \geq 0\} [] \{i = 0\}$

(preserved) ✓

$\{i = 0 \ \& \ i < 0\} [] \{i = 42\}$

(use case) ✓ Why?

$\{i = 0\} [] \text{ while } (i \geq 0) \{\} \{i = 42\}$

Loop Invariant Rule Is Not Sufficient!

Example (Invariant w/o termination validates any postcondition)

$\{i = 0\}$

$[]$

while ($i \geq 0$) $\{\}$

$\{i = 42\}$

Try $i = 0$ as invariant

$(i = 0 \ \& \ i < 0) \leftrightarrow \text{FALSE}$, therefore, **any** Q provable

$\vdash i = 0 \rightarrow i = 0$

(initially valid) ✓

$\{i = 0 \ \& \ i \geq 0\} [] \{i = 0\}$

(preserved) ✓

$\{i = 0 \ \& \ i < 0\} [] \{Q\}$

(use case) ✓

$\frac{}{\{i = 0\} [] \text{while } (i = 0) \{\} \{i = 42\}}$

Loop Invariant Rule Is Not Sufficient!

Example (Invariant w/o termination validates any postcondition)

```
{i = 0}  
[]  
while (i >= 0) {}  
{Q}
```

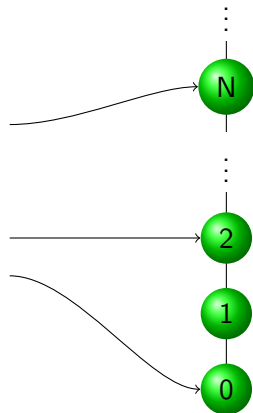
Program does not terminate: this Hoare triple proves nothing

“Ex falso quodlibet”: $\text{FALSE} \rightarrow Q$ is valid formula

Mapping Loop Execution into Well-Founded Order

```
while (b) {  
  body  
}
```

```
if (b) { body }1  
:  
if (b) { body }17  
if (b) { body }42
```



Need to find an expression of type \mathbb{N} getting smaller with each iteration

Such an expression is called a **variant**

The Variant Rule

The Variant

Let *Dec* be a first-order logic integer term, called *variant*

The Variant Rule

The Variant

Let *Dec* be a first-order logic integer term, called **variant**

Invariant Rule

$$\frac{\begin{array}{l} \vdash P \rightarrow \mathcal{U}(\textit{Inv} \quad \quad \quad) \quad \quad \quad (\text{init.} \quad \quad \quad) \\ \{\textit{Inv} \ \& \ b \quad \quad \quad \} \sqcap \pi \{\textit{Inv} \quad \quad \quad \} \quad \quad \quad (\text{pres.} \quad \quad \quad) \\ \quad \quad \quad \{\textit{Inv} \ \& \ !b\} \sqcap \rho \{Q\} \quad \quad \quad (\text{use case}) \end{array}}{\{P\} [\mathcal{U}] \text{ while } (b) \{\pi\} \rho \{Q\}}$$

The Variant Rule

The Variant

Let *Dec* be a first-order logic integer term, called **variant**

Invariant and Variant Rule

$$\frac{\begin{array}{l} \vdash P \rightarrow \mathcal{U}(\textit{Inv} \ \& \ \textit{Dec} \geq 0) \quad \text{(init., positive)} \\ \{ \textit{Inv} \ \& \ b \} \ \Box \ \pi \{ \textit{Inv} \} \quad \text{(pres.)} \\ \{ \textit{Inv} \ \& \ !b \} \ \Box \ \rho \{ Q \} \quad \text{(use case)} \end{array}}{\{ P \} \ [\mathcal{U}] \ \text{while} \ (b) \ \{ \pi \} \ \rho \{ Q \}}$$

The Variant Rule

The Variant

Let Dec be a first-order logic integer term, called **variant**

Invariant and Variant Rule

$$\frac{\begin{array}{l} \vdash P \rightarrow \mathcal{U}(\text{Inv} \ \& \ Dec \geq 0) \quad \text{(init., positive)} \\ \{\text{Inv} \ \& \ b \ \& \ Dec = Dec'\} \sqcap \pi \{\text{Inv} \ \& \ Dec \geq 0 \ \& \ Dec < Dec'\} \quad \text{(pres., decrease)} \\ \{\text{Inv} \ \& \ !b\} \sqcap \rho \{Q\} \quad \text{(use case)} \end{array}}{\{P\} [\mathcal{U}] \text{ while } (b) \{\pi\} \rho \{Q\}}$$

Dec' is new function symbol of type integer

Loop Variant: Example

Example (Countdown)

```
{ n >= 0 }  
[]  
while (n > 0) {  
    n = n - 1;  
}  
{ n = 0 }
```

Loop Variant: Example

Example (Countdown)

```
{ n >= 0 }  
[]  
while (n > 0) {  
    n = n - 1;  
}  
{ n = 0 }
```

Invariant can be $n \geq 0$. What is a suitable variant?

Variant Rule

$$\frac{\begin{array}{l} \vdash P \rightarrow \mathcal{U}(\text{Inv} \ \& \ Dec \geq 0) \quad \text{(init., positive)} \\ \{ \text{Inv} \ \& \ b \ \& \ Dec = Dec' \} \ [] \ \pi \{ \text{Inv} \ \& \ Dec \geq 0 \ \& \ Dec < Dec' \} \quad \text{(pres., decrease)} \\ \{ \text{Inv} \ \& \ !b \} \ [] \ \rho \{ Q \} \quad \text{(use case)} \end{array}}{\{ P \} [\mathcal{U}] \text{ while } (b) \{ \pi \} \rho \{ Q \}}$$

Loop Variant: Example

Example (Unbounded Loop)

$\vdash - n \geq 0 \rightarrow$ $n \geq 0 \ \& \ n \geq 0$	$\left\{ \begin{array}{l} \{n > 0 \ \& \ n = n'\} \\ [] \\ n = n - 1; \\ \{n \geq 0 \ \& \ n < n'\} \end{array} \right.$	$\left\{ \begin{array}{l} \{n \geq 0 \ \& \ n \leq 0\} \\ [] \\ \{n = 0\} \end{array} \right.$
---	--	--

Try n

Variant Rule

$$\frac{\begin{array}{l} \vdash P \rightarrow \mathcal{U}(\text{Inv} \ \& \ Dec \geq 0) \quad (\text{init., positive}) \\ \{\text{Inv} \ \& \ b \ \& \ Dec = Dec'\} \ [] \ \pi \ \{\text{Inv} \ \& \ Dec \geq 0 \ \& \ Dec < Dec'\} \quad (\text{pres., decrease}) \\ \{\text{Inv} \ \& \ !b\} \ [] \ \rho \ \{Q\} \quad (\text{use case}) \end{array}}{\{P\} [\mathcal{U}] \text{ while } (b) \ \{\pi\} \rho \ \{Q\}}$$

Proving Termination in KeY-Hoare

KeY-Hoare Input File Syntax

```
\programVariables {  
    int n;  
}
```

```
\hoareTotal{  
    { n >= 0 }  
    \[ {  
        while (n > 0) {  
            n = n - 1;  
        }  
    } \]  
    { n = 0 }  
}
```

You will be asked for a **variant** as well as for an invariant

Proving Termination Only

If we are interested only in **termination** of a program

Prove total correctness with trivial postcondition **true**

Example (Array Addition, Full Functional Specification)

```
{ len >= 0 &
  \forall int j;
    (j >= 0 & j < len -> a[j] = a0(j) & b[j] = b0(j)) }
i = 0;
while (i < len) {
  a[i] = a[i] + b[i];
  i = i + 1;
}
{ \forall int j;
  (j >= 0 & j < len -> a[j] = a0(j) + b0(j)) }
```

Proving Termination Only

If we are interested only in **termination** of a program

Prove total correctness with trivial postcondition **true**

Example (Array Addition, Termination Only)


```
{ len >= 0 &
  \forall int j;
    (j >= 0 & j < len -> a[j] = a0(j) & b[j] = b0(j)) }
i = 0;
while (i < len) {
  a[i] = a[i] + b[i];
  i = i + 1;
}
{ true }
```

Proving Termination Only

If we are interested only in **termination** of a program

Prove total correctness with trivial postcondition **true**

Example (Array Addition, Termination Only)

```
{ len >= 0 }  Simplified Precondition  
i = 0;  
while (i < len) {  
    a[i] = a[i] + b[i];  
    i = i + 1;  
}  
{ true }
```

Proving Termination Only

If we are interested only in **termination** of a program

Prove total correctness with trivial postcondition **true**

Example (Array Addition, Termination Only)

```
{ len >= 0 } ← Simplified Precondition
i = 0;
while (i < len) {
    a[i] = a[i] + b[i];
    i = i + 1;
}
{ true }
```

Can be proven with nearly trivial invariant $i \geq 0$ and variant $len-i$

Proving Termination Only

If we are interested only in **termination** of a program

Prove total correctness with trivial postcondition **true**

Example (Array Addition, Termination Only)

```
{ len >= 0 } ← Simplified Precondition
i = 0;
while (i < len) {
    a[i] = a[i] + b[i];
    i = i + 1;
}
{ true }
```

Can be proven with nearly trivial invariant $i \geq 0$ and variant $len-i$

Demo arrayAdd.key

Summary

- ▶ The WHILE-language has unbounded value-type arrays
- ▶ In FOL arrays are represented as unary functions
- ▶ During update application, **aliasing analysis** on indices is performed
- ▶ **Partial Correctness** Termination not guaranteed
 Total Correctness Termination whenever precondition holds
- ▶ Proving total correctness: invariant maintained and **variant decreases**
- ▶ Proving **termination only** can be significantly easier