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;
}
```

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 ⇒ 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!

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

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 || a[j] := 2]`

KeY-Hoare introduces conditional expressions to unalias array indices:

`[a[i] := e](a[j])` $\Longrightarrow$ `\if (i=j) \then (e) \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`

- Need to exclude array index aliasing in precondition (`!i=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 $\pi^{\mathcal{U}^s}$.

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 π^s : State $\rightarrow$ 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

Invariant Rule

$$\frac{\begin{array}{cc} \vdash\ P \mathrel{->} \mathcal{U}(\mathit{Inv}) & \text{(initially valid)} \\ \{\mathit{Inv}\ \&\ \texttt{b = true}\}\,[]\,\pi\,\{\mathit{Inv}\} & \text{(preserved)} \\ \{\mathit{Inv}\ \&\ \texttt{b = false}\}\,[]\,\rho\,\{Q\} & \text{(use case)} \end{array}}{\{P\}\,[\mathcal{U}]\,\textbf{while}\ \texttt{(b)}\ \{\pi\}\,\rho\,\{Q\}}$$

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

Invariant Rule, Instantiated

$$\frac{\begin{array}{c}\vdash \texttt{i = 0 -> i = 0} \\ \{\texttt{i = 0 \& i >= 0}\}\,[]\,\{\texttt{i = 0}\} \\ \{\texttt{i = 0 \& i < 0}\}\,[]\,\{\texttt{i = 42}\}\end{array}}{\{\texttt{i = 0}\}\,[]\,\textbf{while}\ \texttt{(i = 0) \{\}}\,\{\texttt{i = 42}\}}$$

(initially valid) ✔
(preserved) ✔
(use case) ✔ Why?

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

`(i = 0 & i < 0) <-> FALSE`, therefore, any Q provable

$$\frac{\begin{array}{c}\vdash \texttt{i = 0} \texttt{ -> } \texttt{i = 0} \\ \{\texttt{i = 0 \& i >= 0}\}\,[]\,\{\texttt{i = 0}\} \\ \{\texttt{i = 0 \& i < 0}\}\,[]\,\{Q\}\end{array}}{\{\texttt{i = 0}\}\,[]\,\textbf{while}\ \texttt{(i = 0)}\ \{\}\,\{\texttt{i = 42}\}}$$

(initially valid) ✔
(preserved) ✔
(use case) ✔

Loop Invariant Rule Is Not Sufficient!

Example (Invariant w/o termination validates any postcondition)

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

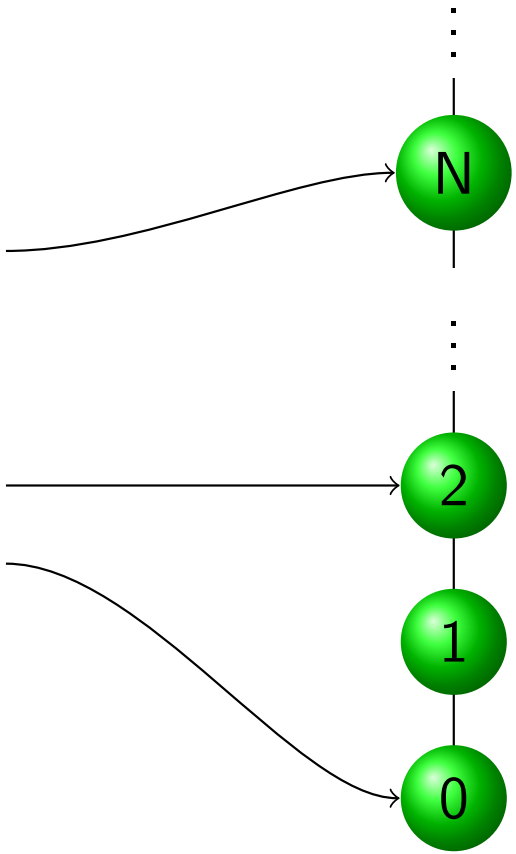
Program does not terminate: this Hoare triple proves nothing

"Ex falso quodlibet": `FALSE -> 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

Invariant and Variant Rule

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

Dec' is new function symbol of type integer

Loop Variant: Example

Example (Countdown)

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

Invariant can be `n >= 0`. What is a suitable variant?

Variant Rule

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

Loop Variant: Example

Example (Unbounded Loop)

`|- n >= 0 ->`	`{n > 0 & n = n'}`	`{n >= 0 & n <= 0}`
`n >= 0 & n >= 0`	`[]`	`[]`
	`n = n - 1;`	
	`{n >= 0 & n < n'}`	`{n = 0}`

Try `n`

Variant Rule

$$\frac{\begin{array}{c}\vdash \; P \mathrel{->} \mathcal{U}(\mathit{Inv} \;\&\; \mathit{Dec} \geq 0) \qquad \text{(init., positive)} \\ \{\mathit{Inv} \;\&\; \mathtt{b} \;\&\; \mathit{Dec}{=}\mathit{Dec}'\}\,[]\,\pi\,\{\mathit{Inv} \;\&\; \mathit{Dec} \geq 0 \;\&\; \mathit{Dec}{<}\mathit{Dec}'\} \qquad \text{(pres., decrease)} \\ \{\mathit{Inv} \;\&\; \mathtt{!b}\}\,[]\,\rho\,\{Q\} \qquad \text{(use case)}\end{array}}{\{P\}\,[\mathcal{U}]\,\mathbf{while}\;\;(\mathtt{b})\;\;\{\pi\}\,\rho\,\{Q\}}$$

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

Simplified Precondition

Can be proven with nearly trivial invariant `i >= 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