Software Engineering using Formal Methods

Reasoning about Programs with Loops and Method Calls

Wolfgang Ahrendt & Richard Bubel (& Reiner Hähnle)
& Wojciech Mostowski

12 October 2011

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement

$$\psi \Longrightarrow \langle \texttt{i=j++;if(isValid) \{ok=true;\}...} \rangle \phi$$

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones

$$\frac{\psi \Longrightarrow \langle \texttt{t=j;j=j+1;i=t;}\textbf{\texttt{if}}\texttt{(isValid) \{ok=}\textbf{\texttt{true}}\texttt{;\}...}\rangle \phi}{\psi \Longrightarrow \langle \texttt{i=j++;}\textbf{\texttt{if}}\texttt{(isValid) \{ok=}\textbf{\texttt{true}}\texttt{;\}...}\rangle \phi}$$

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- atomic assignments to updates

$$\dfrac{\dfrac{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle\mathtt{j=j+1;i=t;}\mathbf{if}\mathtt{(isValid)\ \{ok=}\mathbf{true}\mathtt{;\}\ldots}\rangle\phi}{\psi \Longrightarrow \langle\mathtt{t=j;j=j+1;i=t;}\mathbf{if}\mathtt{(isValid)\ \{ok=}\mathbf{true}\mathtt{;\}\ldots}\rangle\phi}}{\psi \Longrightarrow \langle\mathtt{i=j++;}\mathbf{if}\mathtt{(isValid)\ \{ok=}\mathbf{true}\mathtt{;\}\ldots}\rangle\phi}$$

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- atomic assignments to updates
- accumulated updates capture changed program state

$$\cfrac{\cfrac{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j}+1 \| \mathtt{i} := \mathtt{j}\}\langle \mathtt{\mathbf{if}(isValid)\ \{ok=\mathbf{true};\}\ldots}\rangle\phi}{\ldots}}{\cfrac{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle \mathtt{j=j+1;i=t;\mathbf{if}(isValid)\ \{ok=\mathbf{true};\}\ldots}\rangle\phi}{\cfrac{\psi \Longrightarrow \langle \mathtt{t=j;j=j+1;i=t;\mathbf{if}(isValid)\ \{ok=\mathbf{true};\}\ldots}\rangle\phi}{\psi \Longrightarrow \langle \mathtt{i=j{+}{+};\mathbf{if}(isValid)\ \{ok=\mathbf{true};\}\ldots}\rangle\phi}}}$$

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- atomic assignments to updates
- accumulated updates capture changed program state
- control flow branching induces proof splitting

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\psi, \{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{TRUE}) \Longrightarrow \{\mathcal{U}\}\langle \mathtt{\{ok=\textbf{true};\}}\ldots\rangle\phi \\ \psi, !\{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{FALSE}) \Longrightarrow \{\mathcal{U}\}\langle \ldots\rangle\phi\end{array}}{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j} + 1 \| \mathtt{i} := \mathtt{j}\}\langle \mathtt{\textbf{if}(isValid)\ \{ok=\textbf{true};\}}\ldots\rangle\phi}}{\ldots}}{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle \mathtt{j=j+1;i=t;\textbf{if}(isValid)\ \{ok=\textbf{true};\}}\ldots\rangle\phi}}{\dfrac{\psi \Longrightarrow \langle \mathtt{t=j;j=j+1;i=t;\textbf{if}(isValid)\ \{ok=\textbf{true};\}}\ldots\rangle\phi}{\psi \Longrightarrow \langle \mathtt{i=j\texttt{++};\textbf{if}(isValid)\ \{ok=\textbf{true};\}}\ldots\rangle\phi}}$$

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- atomic assignments to updates
- accumulated updates capture changed program state
- control flow branching induces proof splitting
- application of updates on formula computes weakest precondition

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\psi' \Longrightarrow \{\mathcal{U}'\}\phi \qquad \ldots}{\ldots \qquad \ldots}}{\begin{array}{c}\psi, \{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{TRUE}) \Longrightarrow \{\mathcal{U}\}\langle\{\mathtt{ok{=}}\mathbf{true};\}\ldots\rangle\phi \\ \psi, !\{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{FALSE}) \Longrightarrow \{\mathcal{U}\}\langle\ldots\rangle\phi\end{array}}}{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j}+1 \| \mathtt{i} := \mathtt{j}\}\langle\mathbf{if}(\mathtt{isValid})\ \{\mathtt{ok{=}}\mathbf{true};\}\ldots\rangle\phi}}{\ldots}}{\psi \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle\mathtt{j{=}j{+}1;i{=}t;}\mathbf{if}(\mathtt{isValid})\ \{\mathtt{ok{=}}\mathbf{true};\}\ldots\rangle\phi}}{\dfrac{\psi \Longrightarrow \langle\mathtt{t{=}j;j{=}j{+}1;i{=}t;}\mathbf{if}(\mathtt{isValid})\ \{\mathtt{ok{=}}\mathbf{true};\}\ldots\rangle\phi}{\psi \Longrightarrow \langle\mathtt{i{=}j{+}{+};}\mathbf{if}(\mathtt{isValid})\ \{\mathtt{ok{=}}\mathbf{true};\}\ldots\rangle\phi}}$$

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5?

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable `i` has been increased by 1?

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable `i` has been increased by 1? $\{\mathtt{i} := \mathtt{i+1}\}\phi$
- program variables `i` and `j` swapped values?

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable `i` has been increased by 1? $\{\mathtt{i} := \mathtt{i+1}\}\phi$
- program variables `i` and `j` swapped values? $\{\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{i}\}\phi$
- all components of an array `arr` of length 2 have value 0?

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable i has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable i has been increased by 1? $\{\mathtt{i} := \mathtt{i+1}\}\phi$
- program variables i and j swapped values? $\{\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{i}\}\phi$
- all components of an array arr of length 2 have value 0?
  $\{\mathtt{arr[0]} := 0 \parallel \mathtt{arr[1]} := 0\}\phi$
- all components of an array arr of length n have value 0?

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable `i` has been increased by 1? $\{\mathtt{i} := \mathtt{i+1}\}\phi$
- program variables `i` and `j` swapped values? $\{\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{i}\}\phi$
- all components of an array `arr` of length 2 have value 0?
  $$\{\mathtt{arr[0]} := 0 \parallel \mathtt{arr[1]} := 0\}\phi$$
- all components of an array `arr` of length `n` have value 0?

For example to deal with things like

$$\langle \mathtt{\textbf{int}[]\ a\ =\ \textbf{new}\ \textbf{int}[n];} \rangle$$
$$\forall\, \mathtt{\textbf{int}}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 0)$$

Quantified Updates

Definition (Quantified Update)

For T well-ordered type (no ∞ descending chains): quantified update:

$$\{\texttt{\textbackslash for}\ T\ x; \texttt{\textbackslash if}\ \phi(x); l(x) := r(x)\}$$

- For all objects d in T such that $\phi(d)$ perform the updates $\{l(d) := r(d)\}$ in parallel
- If there are several l with conflicting d then choose T-minimal one

- The conditional expression is optional
- Typically, x occurs in ϕ, l, and r (but doesn't need to)
- There is a normal form for updates computed efficiently by KeY

Quantified Updates Cont'd

Example (Initialization of field `a` for all objects in class `C`)

$$\{\texttt{\textbf{\textbackslash for}}\ \mathtt{C}\ o; \mathtt{o.a} := 0\}$$

Quantified Updates Cont'd

Example (Initialization of field `a` for all objects in class `C`)

$$\{\texttt{\textbackslash for}\ \mathtt{C}\ o; \mathrm{o.a} := 0\}$$

Example (Initialization of components of array `a`)

$$\{\texttt{\textbackslash for int}\ i; \mathrm{a}[\mathtt{i}] := 0\}$$

Quantified Updates Cont'd

Example (Initialization of field `a` for all objects in class `C`)

$$\{\texttt{\textbackslash for}\ \mathtt{C}\ o; \mathrm{o.a} := 0\}$$

Example (Initialization of components of array `a`)

$$\{\texttt{\textbackslash for}\ \texttt{int}\ i; \mathrm{a[i]} := 0\}$$

Example (Integer types are well-ordered in KeY)

$$\{\texttt{\textbackslash for}\ \texttt{int}\ i; \mathrm{a[0]} := i\}(\mathrm{a[0]} \doteq 0)$$

- Non-standard order for $\mathbb{Z}$ (with 0 smallest and preserving $<$ for arguments of same sign)
- Proven automatically by update simplifier

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{if}(b) \{ p; \textbf{while}(b) p\}}\, \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{while}(b) p}\, \omega]\phi, \Delta}$$

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{\textbf{if}(b) \{ p; \textbf{while}(b) p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{\textbf{while}(b) p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations?

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{if}(b) \{ p; \textbf{while}(b) p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{while}(b) p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{\textbf{if}(b) \{ p; \textbf{while}(b) p\}}\,\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{\textbf{while}(b) p}\,\omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations?

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{\textbf{if}(b)\ \{\ p;\ \textbf{while}(b)\ p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{\textbf{while}(b)\ p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{if (b) \{ p; while (b) p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{while (b) p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$
- 10000 iterations?

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{if(b) \{ p; while(b) p\}}\,\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{while(b) p}\,\omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind 1×
- 10 iterations? Unwind 11×
- 10000 iterations? Unwind 10001×
  (and don't make any plans for the rest of the day)

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{if(b) \{ p; while(b) p\}}\,\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{while(b) p}\,\omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind 1×
- 10 iterations? Unwind 11×
- 10000 iterations? Unwind 10001×
  (and don't make any plans for the rest of the day)
- an unknown number of iterations?

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{if}\,(\texttt{b})\ \{\ \texttt{p};\ \texttt{while}\,(\texttt{b})\ \texttt{p}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\texttt{while}\,(\texttt{b})\ \texttt{p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$
- 10000 iterations? Unwind $10001\times$
  (and don't make any plans for the rest of the day)
- an unknown number of iterations?

We need an invariant rule (or some other form of induction)

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Encode the desired postcondition after loop into *Inv*

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Encode the desired postcondition after loop into *Inv*

Basic Invariant Rule

loopInvariant

$$\Gamma \Longrightarrow \mathcal{U}[\pi\,\mathtt{while(b)\ p}\ \omega]\phi, \Delta$$

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Encode the desired postcondition after loop into *Inv*

Basic Invariant Rule

$$\Gamma \Longrightarrow \mathcal{U}\,\mathit{Inv}, \Delta \qquad \text{(initially valid)}$$

loopInvariant

$$\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{\textbf{while}}\,\mathtt{(b)}\ \mathtt{p}\ \omega]\phi, \Delta$$

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Encode the desired postcondition after loop into *Inv*

Basic Invariant Rule

loopInvariant

$$\frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\,\mathit{Inv}, \Delta & \text{(initially valid)} \\ \mathit{Inv},\; b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\mathtt{while}(\mathtt{b})\;\mathtt{p}\;\omega]\phi, \Delta}$$

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Encode the desired postcondition after loop into *Inv*

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\,\mathit{Inv}, \Delta & \text{(initially valid)} \\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved)} \\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while}\,\mathtt{(b)}\ \mathtt{p}\ \omega]\phi, \Delta}$$

Loop Invariants Cont'd

Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved)}\\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while(b)\ p}\ \omega]\phi, \Delta}$$

Loop Invariants Cont'd

Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved)}\\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{while}(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise:

  Γ, Δ in general don't hold in state defined by $\mathcal{U}$

  2nd premise *Inv* must be invariant for any state, not only $\mathcal{U}$

  3rd premise We don't know the state after the loop exits

Loop Invariants Cont'd

Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved)}\\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while}(\mathrm{b})\ \mathrm{p}\ \omega]\phi, \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise:

  Γ, Δ in general don't hold in state defined by $\mathcal{U}$

  2nd premise *Inv* must be invariant for any state, not only $\mathcal{U}$

  3rd premise We don't know the state after the loop exits

- But: context contains (part of) precondition and class invariants

Loop Invariants Cont'd

Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathtt{p}]\mathit{Inv} & \text{(preserved)}\\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathbf{while}(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise:

  Γ, Δ in general don't hold in state defined by $\mathcal{U}$

  2nd premise *Inv* must be invariant for any state, not only $\mathcal{U}$

  3rd premise We don't know the state after the loop exits
- But: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

Example

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Example

Precondition: $a \doteq\!\!\!\!/\ \texttt{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Example

Precondition: $\mathtt{a} \doteq\!\!\!\!/\ \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Example

Precondition: $a \neq$ **null**

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall$ **int** x; $(0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \texttt{i}$ & $\texttt{i} \leq \texttt{a.length}$

Example

Precondition: $\mathtt{a} \doteq\!\!\!\!/\ \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$

Example

Precondition: $\texttt{a} \neq \texttt{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \texttt{int}\ x;\ (0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \texttt{i} \ \&\ \texttt{i} \leq \texttt{a.length}$
$\&\ \forall\, \texttt{int}\ x;\ (0 \leq x < \texttt{i} \rightarrow \texttt{a}[x] \doteq 1)$
$\&\ \texttt{a} \neq \texttt{null}$

Example

Precondition: $\mathtt{a} \doteq\!\!\!\!/\; \mathbf{null}$ & *ClassInv*

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$
$\&\ \ \mathtt{a} \not\doteq \mathbf{null}$
& *ClassInv'*

Keeping the Context

- Want to keep part of the context that is unmodified by loop

Keeping the Context

- Want to keep part of the context that is unmodified by loop
- **`assignable`** clauses for loops can tell what might be modified

```
@ assignable i, a[*];
```

Keeping the Context

- Want to keep part of the context that is unmodified by loop
- **`assignable`** clauses for loops can tell what might be modified

```
@ assignable i, a[*];
```

- How to erase all values of **`assignable`** locations in formula Γ ?

Keeping the Context

- Want to keep part of the context that is unmodified by loop
- **assignable** clauses for loops can tell what might be modified

```
@ assignable i, a[*];
```

- How to erase all values of **assignable** locations in formula Γ ?

  Analogous situation: $\forall$-Right quantifier rule $\Longrightarrow \forall x; \phi$
  Replace x with a fresh constant $*$

  To change value of program location use update, not substitution

Keeping the Context

- Want to keep part of the context that is unmodified by loop
- `assignable` clauses for loops can tell what might be modified

```
@ assignable i, a[*];
```

- How to erase all values of `assignable` locations in formula Γ ?

  Analogous situation: $\forall$-Right quantifier rule $\Longrightarrow \forall\, x;\ \phi$
  Replace x with a fresh constant $*$

  To change value of program location use update, not substitution
- Anonymising updates $\mathcal{V}$ erase information about modified locations

$\mathcal{V} \ = \ \{\mathtt{i} := \mathtt{c} \;||\; \mathtt{\backslash for}\ x;\ \mathtt{a}[x] := f_a(x)\}$
(c, f_a new constant resp. function symbol)

Loop Invariants Cont'd

Improved Invariant Rule

$$\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{while(b) p}\ \omega]\phi, \Delta$$

Loop Invariants Cont'd

Improved Invariant Rule

$$\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta \qquad \text{(initially valid)}$$

$$\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{while(b) p}\ \omega]\phi, \Delta$$

Loop Invariants Cont'd

Improved Invariant Rule

$$\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta \qquad \text{(initially valid)}$$

$$\Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv} \;\&\; b \doteq \mathtt{TRUE} \rightarrow [\mathrm{p}]\mathit{Inv}), \Delta \qquad \text{(preserved)}$$

$$\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{\textbf{while}(b)\ p}\ \omega]\phi, \Delta$$

Loop Invariants Cont'd

Improved Invariant Rule

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{TRUE} \rightarrow [\mathtt{p}]\mathit{Inv}), \Delta & \text{(preserved)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{\textbf{while}(b)\ p}\ \omega]\phi, \Delta}$$

Loop Invariants Cont'd

Improved Invariant Rule

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{TRUE} \rightarrow [\mathtt{p}]\mathit{Inv}), \Delta & \text{(preserved)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{\textbf{while}(b)\ p}\ \omega]\phi, \Delta}$$

- Context is kept as far as possible
- Invariant does not need to include unmodified locations
- For **`assignable \everything`** (the default):
  - $\mathcal{V} = \{* := *\}$ wipes out **all** information
  - Equivalent to basic invariant rule
  - Avoid this! Always give a specific **`assignable`** clause

Example with Improved Invariant Rule

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Example with Improved Invariant Rule

Precondition: a $\dot{\neq}$ **null**

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Example with Improved Invariant Rule

Precondition: $\texttt{a} \doteq\!\!\!\!/\ \textbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \textbf{int}\ x;\ (0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

Example with Improved Invariant Rule

Precondition: $\mathtt{a} \doteq\!\!\!\!/\ \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\,\mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$

Example with Improved Invariant Rule

Precondition: $\mathtt{a} \doteq\!\!\!\!/\; \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$

Example with Improved Invariant Rule

Precondition: $\mathtt{a} \doteq\!\!\!\!/\ \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$

Example with Improved Invariant Rule

Precondition: $\texttt{a} \doteq\!\!\!\!/\ \textbf{null}$ & *ClassInv*

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \textbf{int}\ x;\ (0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \texttt{i}\ \ \&\ \ \texttt{i} \leq \texttt{a.length}$
$\&\ \ \forall\, \textbf{int}\ x;\ (0 \leq x < \texttt{i} \rightarrow \texttt{a}[x] \doteq 1)$

Example in JML/Java – Loop.java

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true;
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  (0 <= i && i <= a.length &&
    @   (\forall int x; 0<=x && x<i; a[x]==1));
    @ assignable i, a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

Example from previous lectures

$\forall$ `int` x;
$(x \doteq \mathtt{n} \land x >= 0 \rightarrow$
$[$
```
i = 0; r = 0;
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```
$]\mathtt{r} \doteq ?)$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Example from previous lectures

$\forall$ `int` x;
$(x \doteq \mathtt{n} \land x >= 0 \rightarrow$
$[$

```
i = 0; r = 0;
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```

$]\mathtt{r} \doteq x * x)$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Example from previous lectures

$\forall$ `int` x;
$(x \doteq \mathtt{n} \land x >= 0 \rightarrow$
$[$
```
i = 0; r = 0;
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```
$]\mathtt{r} \doteq x * x)$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Solution:

```
@ loop_invariant
@     i>=0 && 2*r == i*(i + 1) && i <= n;
@ assignable i, r;
```

Example from previous lectures

$\forall$ int x;
  $(x \doteq \mathtt{n} \land x >= 0 \rightarrow$
    $[$ `i = 0; r = 0;`
```
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```
  $]\mathtt{r} \doteq x * x)$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Solution:

```
@ loop_invariant
@     i>=0 && 2*r == i*(i + 1) && i <= n;
@ assignable i, r;
```

File: Loop2.java

Hints

Proving `assignable`

- The invariant rule assumes that **`assignable`** is correct
  E.g., with **`assignable \nothing;`** one can prove nonsense
- Invariant rule of KeY generates proof obligation that ensures correctness of **`assignable`**

Hints

Proving `assignable`

- The invariant rule assumes that **`assignable`** is correct
  E.g., with **`assignable \nothing;`** one can prove nonsense
- Invariant rule of KeY generates proof obligation that ensures correctness of **`assignable`**

Setting in the KeY Prover when proving loops

- Loop treatment: Invariant
- Quantifier treatment: No Splits with Progs
- If program contains `*`, `/`:
  Arithmetic treatment: DefOps
- Is search limit high enough (time out, rule apps.)?
- When proving partial correctness, add **`diverges true;`**

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;`
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;`
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Example (The `array` loop)

```
@ decreasing
```

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;`
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Example (The `array` loop)

```
@ decreasing  a.length - i;
```

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;`
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Example (The `array` loop)

```
@ decreasing  a.length - i;
```

Files:

- `LoopT.java`
- `Loop2T.java`

Method Calls – Repetition

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \ldots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where `m` declared as **`void`** $\mathtt{m}(\mathtt{T}_0\ \mathtt{p}_0, \ldots, \mathtt{T}_n\ \mathtt{p}_n)$

Actions of rule methodCall

- for each formal parameter $\mathtt{p}_\mathtt{i}$ of `m`:
  declare and initialize new local variable $\mathtt{T}_\mathtt{i}\ \mathtt{p\#i} = arg_i;$

Method Calls – Repetition

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \ldots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where $\mathtt{m}$ declared as **void** $\mathtt{m}(\mathtt{T_0}\ \mathtt{p_0}, \ldots, \mathtt{T_n}\ \mathtt{p_n})$

Actions of rule methodCall

- for each formal parameter $\mathtt{p_i}$ of $\mathtt{m}$:
  declare and initialize new local variable $\mathtt{T_i}\ \mathtt{p\#i} = arg_i;$
- look up implementation class C of $\mathtt{m}$ and split proof if implementation cannot be uniquely determined

Method Calls – Repetition

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \ldots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where $\mathtt{m}$ declared as $\mathtt{\textbf{void}}\ \mathtt{m}(\mathtt{T_0}\ \mathtt{p_0}, \ldots, \mathtt{T_n}\ \mathtt{p_n})$

Actions of rule methodCall

- for each formal parameter $\mathtt{p_i}$ of $\mathtt{m}$:
  declare and initialize new local variable $\mathtt{T_i}\ \mathtt{p\#i} = arg_i;$
- look up implementation class C of $\mathtt{m}$ and split proof
  if implementation cannot be uniquely determined
- create method invocation $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})\mathtt{@}C$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_\texttt{i}\ \texttt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=c)\{}\ \texttt{body}\ \texttt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#0,...,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_\texttt{i}\ \texttt{p\#i} = arg_i$;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=c)\{ body \}}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#0,...,p\#n)@C;}\ \omega \rangle \phi, \Delta}$$

Symbolic Execution

Only static information available, proof splitting;

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\mathtt{T_i\ p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \mathtt{method\text{-}frame(source=C,\ this=c)\{\ body\ \}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \mathtt{c.m(p\#0,\ldots,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Symbolic Execution

Only static information available, proof splitting;
Runtime infrastructure required in calculus

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_\texttt{i}\ \texttt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=c)\{ body \}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#0,...,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Symbolic Execution

Only static information available, proof splitting;
Runtime infrastructure required in calculus

File: `inlineDynamicDispatch.key`

Problem

Formal specification of Java API and other called methods

How to perform symbolic execution when Java API method is called?

1. Method has reference implementation in Java
   Inline method body and execute symbolically

Problem

Formal specification of JAVA API and other called methods

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA
   Inline method body and execute symbolically

   Problems Reference implementation not always available

Problem

Formal specification of JAVA API and other called methods

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA
   Inline method body and execute symbolically

   Problems Reference implementation not always available
   Too expensive

Problem

Formal specification of JAVA API and other called methods

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA
   Inline method body and execute symbolically

   Problems Reference implementation not always available
   Too expensive
   Impossible to deal with recursion

Problem

Formal specification of JAVA API and other called methods

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA
   Inline method body and execute symbolically

   Problems Reference implementation not always available
   Too expensive
   Impossible to deal with recursion

2. Use method contract instead of method implementation

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

$$\frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\texttt{preNormal}), \Delta \quad \text{(precondition)}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})\ p\ \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{preNormal}), \Delta & \text{(precondition)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod}(\mathcal{F}(\mathtt{normalPost}) \to \langle \pi\, \mathrm{p}\, \omega \rangle \phi), \Delta & \text{(normal)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_n)\; p\; \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires preExc;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires preExc;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

$$\frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\texttt{preExc}), \Delta \quad \text{(precondition)}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})\; p\; \omega\rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires preExc;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{preExc}), \Delta \quad \text{(precondition)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod}((\mathtt{exc} \neq \mathbf{null} \land \mathcal{F}(\mathtt{excPost})) \\ \qquad\qquad\qquad \rightarrow \langle \pi\, \mathbf{throw}\ \mathtt{exc;}\ \mathrm{p}\, \omega \rangle \phi), \Delta \quad \text{(exceptional)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_n)\ p\, \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$((\mathtt{exc} \doteq \mathbf{null} \wedge \mathcal{F}(\mathtt{\textbf{\textbackslash old}}(\mathtt{preNormal})) \rightarrow \mathcal{F}(\mathit{normalPost})) \wedge$$
$$((\mathtt{exc} \not\doteq \mathbf{null} \wedge \mathcal{F}(\mathtt{\textbf{\textbackslash old}}(\mathtt{preExc}))) \rightarrow \mathcal{F}(\mathit{excPost}))$$

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$((\mathtt{exc} \doteq \mathbf{null} \land \mathcal{F}(\mathbf{\backslash old}(\mathtt{preNormal})) \rightarrow \mathcal{F}(\mathit{normalPost})) \land$$
$$((\mathtt{exc} \not\doteq \mathbf{null} \land \mathcal{F}(\mathbf{\backslash old}(\mathtt{preExc}))) \rightarrow \mathcal{F}(\mathit{excPost}))$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\mathtt{preNormal}) \lor \mathcal{F}(\mathtt{preExc})), \Delta \quad \text{(precondition)}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n})\; p\; \omega\rangle\phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$((\texttt{exc} \doteq \textbf{null} \wedge \mathcal{F}(\texttt{\textbackslash old}(\texttt{preNormal})) \rightarrow \mathcal{F}(normalPost)) \wedge$$
$$((\texttt{exc} \not\doteq \textbf{null} \wedge \mathcal{F}(\texttt{\textbackslash old}(\texttt{preExc}))) \rightarrow \mathcal{F}(excPost))$$

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{preNormal}) \vee \mathcal{F}(\texttt{preExc})), \Delta & \text{(precondition)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{normal}}((\texttt{exc} \doteq \textbf{null} \wedge \phi_{post}) \rightarrow \langle \pi\, \texttt{p}\, \omega \rangle \phi), \Delta & \text{(normal)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})\; p\; \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$((\mathtt{exc} \doteq \mathbf{null} \wedge \mathcal{F}(\backslash\mathtt{old}(\mathtt{preNormal})) \rightarrow \mathcal{F}(\mathit{normalPost})) \wedge$$
$$((\mathtt{exc} \not\doteq \mathbf{null} \wedge \mathcal{F}(\backslash\mathtt{old}(\mathtt{preExc}))) \rightarrow \mathcal{F}(\mathit{excPost}))$$

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\mathtt{preNormal}) \vee \mathcal{F}(\mathtt{preExc})), \Delta & \text{(precondition)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{normal}}((\mathtt{exc} \doteq \mathbf{null} \wedge \phi_{post}) \rightarrow \langle \pi\, \mathtt{p}\, \omega \rangle \phi), \Delta & \text{(normal)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{exc}}((\mathtt{exc} \not\doteq \mathbf{null} \wedge \phi_{post}) & \\ \qquad \rightarrow \langle \pi\, \mathbf{throw}\ \mathtt{exc};\ \mathtt{p}\, \omega \rangle \phi), \Delta & \text{(exceptional)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_n)\ p\, \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Understanding Proof Situations

Reasons why a proof may not close

- bug or incomplete specification
- bug in program
- maximal number of steps reached: restart or increase # of steps
- automatic proof search fails and manual rule applications necessary

Understanding Proof Situations

Reasons why a proof may not close

- bug or incomplete specification
- bug in program
- maximal number of steps reached: restart or increase # of steps
- automatic proof search fails and manual rule applications necessary

Understanding open proof goals

- follow the taken control-flow from the root to the open goal
- branch labels may give useful hints
- identify (part of) the post-condition or invariant that cannot be proven
- sequent remains always in "pre-state".
  I.e., constraints like $\mathtt{i} \geq 0$ refer to the value of $\mathtt{i}$ before executing the program (exception: formula is behind update or modality)
- remember: $\Gamma \Longrightarrow o \doteq \mathbf{null}, \Delta$ is equivalent to $\Gamma, o \not\doteq \mathbf{null} \Longrightarrow \Delta$

Summary

- Most JAVA features covered in KeY
- Several of remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is high
- Proving loops requires user to provide invariant
  - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic

Literature for this Lecture

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.6.7, 3.7