Formale Entwicklung objektorientierter Software

Praktikum im Wintersemester 2007/2008

Prof. P. H. Schmitt, Dr. T. Käufl, C. Engel, B. Weiß

Institut für Theoretische Informatik
Universität Karlsruhe

12. Dezember 2007

Loop Invariants

Unwinding Loops

$$\text{loopUnwind} \; \frac{}{\Gamma \; \texttt{==>} \; \mathcal{U}\langle \pi \; \texttt{while(e) p} \; \omega \rangle \phi, \; \Delta}$$

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

How to handle a loop with...

- 0 iterations?

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$.

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

How to handle a loop with...

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

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{if(e)\{p while(e) p\}}\ \omega\rangle\psi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

How to handle a loop with...

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

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

How to handle a loop with...

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

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{if(e)\{p while(e) p\}}\ \omega\rangle\psi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

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).

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{if(e)\{p while(e) p\}}\ \omega\rangle\psi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

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?

Unwinding Loops

$$\text{loopUnwind}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{if(e)}\{\texttt{p while(e) p}\}\ \omega\rangle\psi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{while(e) p}\ \omega\rangle\phi,\ \Delta}$$

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?

$\leadsto$ We need an **invariant rule**. (or some other form of induction)

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta \qquad \text{(initially valid)}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)} \\ Inv, \texttt{e} \texttt{ ==> } [\texttt{p}] Inv & \text{(preserved)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Basic Invariant Rule

$$
\text{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)} \\ Inv, \texttt{e ==> } [\texttt{p}] Inv & \text{(preserved)} \\ Inv, \texttt{!e ==> } [\pi\ \omega]\phi & \text{(use case)} \end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}
$$

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)}\\ Inv, \texttt{e} \texttt{ ==> } [\mathrm{p}] Inv & \text{(preserved)}\\ Inv, \texttt{!e} \texttt{ ==> } [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

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

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U}Inv,\ \Delta & \text{(initially valid)}\\ Inv, \texttt{e} \texttt{ ==> } [\texttt{p}]Inv & \text{(preserved)}\\ Inv, \texttt{!e} \texttt{ ==> } [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premiss
- Context contains (parts of) precondition and class invariants

Basic Invariant Rule

$$
\text{loopInvariant}\;\frac{\begin{array}{ll} \Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)} \\ Inv, \texttt{e ==> } [\texttt{p}] Inv & \text{(preserved)} \\ Inv, \texttt{!e ==> } [\pi\ \omega]\phi & \text{(use case)} \end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}
$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premiss
- Context contains (parts 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] = 0;
    i++;
}
```

Example

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

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

Example

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

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

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

Example

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

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

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

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

Example

Precondition: $\texttt{a} \not\doteq \texttt{null}$

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

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

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

Example

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

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

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

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

Example

Precondition: $\mathtt{a} \not\doteq \mathtt{null}$ & $\mathit{ClassInv}$

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

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

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

Keeping the Context

- We would like to keep unmodified parts of the context

Keeping the Context

- We would like to keep unmodified parts of the context
- **`assignable` clauses** for loops can tell what may be modified

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

Keeping the Context

- We would like to keep unmodified parts of the context
- **`assignable` clauses** for loops can tell what may be modified

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

- But: determining unaffected formulas syntactically is impossible because of aliasing

Keeping the Context

- We would like to keep unmodified parts of the context
- **`assignable` clauses** for loops can tell what may be modified

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

- But: determining unaffected formulas syntactically is impossible because of aliasing
- Solution: **anonymising updates** $\mathcal{V}$ delete information about modified locations

$$\mathcal{V} \;=\; \{\mathtt{i} := i_0 \;\mid\; \mathtt{for}\ x;\ \mathtt{a}[x] := arr_0(\mathtt{a}, x)\}$$

Improved Invariant Rule

loopInvariant $\dfrac{}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}Inv,\ \Delta \qquad \text{(initially valid)}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U}Inv,\ \Delta & \text{(initially valid)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{!e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv\ \texttt{\&}\ \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv\ \texttt{\&}\ \texttt{!e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

- Context is kept as far as possible

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)} \\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)} \\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{!e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

- Context is kept as far as possible
- Invariant does not need to talk about unmodified locations

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma\ \texttt{==>}\ \mathcal{U}Inv,\ \Delta & \text{(initially valid)}\\ \Gamma\ \texttt{==>}\ \mathcal{U}\mathcal{V}(Inv\ \texttt{\&}\ \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\\ \Gamma\ \texttt{==>}\ \mathcal{U}\mathcal{V}(Inv\ \texttt{\&}\ \texttt{!e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma\ \texttt{==>}\ \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

- Context is kept as far as possible
- Invariant does not need to talk about unmodified locations
- For `assignable \everything` (the default):
  - $\mathcal{V} = \{* := *\}$ wipes out **all** information

Improved Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U} Inv,\ \Delta & \text{(initially valid)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& !e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}$$

- Context is kept as far as possible
- Invariant does not need to talk about unmodified locations
- For `assignable \everything` (the default):
  - $\mathcal{V} = \{* := *\}$ wipes out **all** information
  - Equivalent to basic invariant rule

Improved Invariant Rule

$$
\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U}Inv,\ \Delta & \text{(initially valid)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{e} \rightarrow [\texttt{p}]Inv),\ \Delta & \text{(preserved)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(Inv \texttt{ \& } \texttt{!e} \rightarrow [\pi\ \omega]\phi),\ \Delta & \text{(use case)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}[\pi\ \texttt{while(e) p;}\ \omega]\phi,\ \Delta}
$$

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

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Proof obligation

$$\mathcal{V} anon \;\rightarrow\; [\mathtt{p}]\mathcal{V} anon$$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Proof obligation

$$\mathcal{V} anon \;\rightarrow\; [\mathtt{p}]\mathcal{V} anon$$

Example

$$\{\mathtt{i} := i_0\} anon \;\rightarrow\; [\mathtt{i=5;}]\{\mathtt{i} := i_0\} anon$$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Proof obligation

$$\mathcal{V}anon \;\rightarrow\; [\mathtt{p}]\mathcal{V}anon$$

Example

$$\frac{\{\mathtt{i} := i_0\}anon \;\rightarrow\; \{\mathtt{i} := 5 \mid \mathtt{i} := i_0\}anon}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; [\mathtt{i=5;}]\{\mathtt{i} := i_0\}anon}$$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use “heap-dependent” predicate $anon$

Proof obligation

$$\mathcal{V}anon \;\rightarrow\; [\mathtt{p}]\mathcal{V}anon$$

Example

$$\cfrac{\cfrac{\{\mathtt{i} := i_0\}anon \;\rightarrow\; \{\mathtt{i} := i_0\}anon}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; \{\mathtt{i} := 5 \mid \mathtt{i} := i_0\}anon}}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; [\mathtt{i=5;}]\{\mathtt{i} := i_0\}anon}$$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Proof obligation

$$\mathcal{V}anon \;\rightarrow\; [\mathtt{p}]\mathcal{V}anon$$

Example

$$\dfrac{\dfrac{\dfrac{*}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; \{\mathtt{i} := i_0\}anon}}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; \{\mathtt{i} := 5 \mid \mathtt{i} := i_0\}anon}}{\{\mathtt{i} := i_0\}anon \;\rightarrow\; [\mathtt{i=5;}]\{\mathtt{i} := i_0\}anon}$$

Addition 1: Proving `assignable`

- The invariant rule **assumes** the `assignable` is correct
- For soundness, we need to **prove** this
- Solution: Use "heap-dependent" predicate $anon$

Proof obligation

$$\mathcal{V} anon \ \rightarrow \ [\mathtt{p}]\mathcal{V} anon$$

Example

$$\dfrac{\dfrac{\dfrac{*}{\{\mathtt{i} := i_0\}anon \ \rightarrow \ \{\mathtt{i} := i_0\}anon}}{\{\mathtt{i} := i_0\}anon \ \rightarrow \ \{\mathtt{i} := 5 \mid \mathtt{i} := i_0\}anon}}{\{\mathtt{i} := i_0\}anon \ \rightarrow \ [\mathtt{i=5;}]\{\mathtt{i} := i_0\}anon}$$

$\leadsto$ Add $\mathcal{V} anon$ to Inv in "preserved" case.

Addition 2: Proving Termination

- The invariant rule only proves partial correctness

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

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

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

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

$\boxed{v = \texttt{a.length-i}}$

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

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

$\boxed{v = \texttt{a.length-i}}$

$\leadsto$ - Add $v \geq 0$ to Inv

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

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

$\boxed{v = \texttt{a.length-i}}$

$\leadsto$ - Add $v \geq 0$ to Inv

- Add $v < v^{old}$ to right of "preserved" case

Addition 2: Proving Termination

- The invariant rule only proves partial correctness
- Solution: Find a decreasing integer term v (called **variant**)
  - $v \geq 0$ is initially valid
  - $v \geq 0$ is preserved by the loop body
  - v is strictly decreased by the loop body

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

$\boxed{v = \texttt{a.length-i}}$

$\rightsquigarrow$ - Add $v \geq 0$ to Inv

- Add $v < v^{old}$ to right of "preserved" case

- Replace box with diamond

Further Complications

Since Java is a **real** language...

- the loop guard expression e may have side effects

Further Complications

Since Java is a **real** language...

- the loop guard expression e may have side effects
- both e and the loop body may throw an exception

Further Complications

Since Java is a **real** language...

- the loop guard expression `e` may have side effects
- both `e` and the loop body may throw an exception
- the loop body may use `break` or `continue`

Further Complications

Since Java is a **real** language...

- the loop guard expression `e` may have side effects
- both `e` and the loop body may throw an exception
- the loop body may use `break` or `continue`

$\leadsto$ The invariant rule in KeY takes all this into account.

Loop Specifications in JML

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

Loop Specifications in JML

```
int i = 0;
/*@ loop_invariant
  @   0<= i && i<=a.length
  @   && (\forall int x; 0<=x && x<a.length; a[x]==0);
  @
  @
  @*/
while(i < a.length) {
    a[i] = 0;
    i++;
}
```

Loop Specifications in JML

```
int i = 0;
/*@ loop_invariant
  @   0<= i && i<=a.length
  @   && (\forall int x; 0<=x && x<a.length; a[x]==0);
  @ assignable i, a[*];
  @
  @*/
while(i < a.length) {
    a[i] = 0;
    i++;
}
```

Loop Specifications in JML

```
int i = 0;
/*@ loop_invariant
  @   0<= i && i<=a.length
  @   && (\forall int x; 0<=x && x<a.length; a[x]==0);
  @ assignable i, a[*];
  @ decreases a.length-i;
  @*/
while(i < a.length) {
    a[i] = 0;
    i++;
}
```

Using Method Contracts

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle \pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle \pi\ \texttt{method-frame(C,o)}\{\texttt{b}\}\ \omega\rangle\phi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle \pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{method-frame(C,o)}\{\texttt{b}\}\ \omega\rangle\phi,\ \Delta}{\Gamma\ \texttt{==>}\ \mathcal{U}\langle\pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Disadvantages:

- Non-modular

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle \pi\ \texttt{method-frame(C,o)}\{\texttt{b}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle \pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Disadvantages:

- Non-modular
- Duplication of effort

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{method-frame(C,o)}\{\texttt{b}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Disadvantages:

- Non-modular
- Duplication of effort
- Can lead to huge proofs

Expanding Method Bodies

$$\text{methodBodyExpand}\ \frac{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{method-frame(C,o)}\{\texttt{b}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Disadvantages:

- Non-modular
- Duplication of effort
- Can lead to huge proofs
- Sometimes the method body is not available (e.g. `native` methods)

Use Method Contract Rule

Given a contract (Pre, $Post$, $\mathcal{V}$):

Use Method Contract Rule

Given a contract (Pre, $Post$, $\mathcal{V}$):

$$\frac{}{\Gamma \texttt{ ==> } \mathcal{U}\langle \pi \texttt{ o.m(p}_1\texttt{, ..., p}_n\texttt{)@C; } \omega\rangle\phi,\ \Delta}$$

Use Method Contract Rule

Given a contract $(Pre, Post, \mathcal{V})$:

$$\frac{\Gamma \texttt{ ==> } \mathcal{U} Pre,\ \Delta \qquad \text{(Pre)}}{\Gamma \texttt{ ==> } \mathcal{U}\langle \pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Use Method Contract Rule

Given a contract (Pre, $Post$, $\mathcal{V}$):

$$\frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U}Pre,\ \Delta & \text{(Pre)} \\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(\texttt{exc} \doteq \texttt{null} \land Post \rightarrow \langle\pi\ \omega\rangle\phi),\ \Delta & \text{(Post)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

Use Method Contract Rule

Given a contract $(Pre, Post, \mathcal{V})$:

$$\frac{\begin{array}{ll}\Gamma \texttt{ ==> } \mathcal{U}Pre,\ \Delta & \text{(Pre)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(\texttt{exc} \doteq \texttt{null} \land Post \rightarrow \langle\pi\ \omega\rangle\phi),\ \Delta & \text{(Post)}\\ \Gamma \texttt{ ==> } \mathcal{U}\mathcal{V}(\texttt{exc} \not\doteq \texttt{null} \land Post \rightarrow \langle\pi\ \texttt{throw exc;}\ \omega\rangle\phi),\ \Delta & \text{(Exc.)}\end{array}}{\Gamma \texttt{ ==> } \mathcal{U}\langle\pi\ \texttt{o.m(p}_1\texttt{, ..., p}_n\texttt{)@C;}\ \omega\rangle\phi,\ \Delta}$$

THE

THE
END