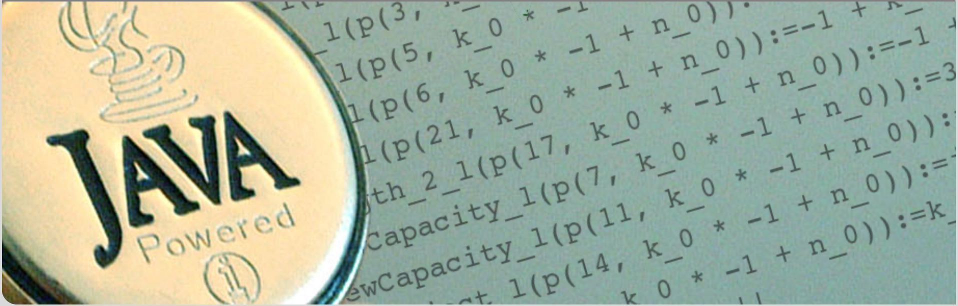
Karlsruhe Institute of Technology

Applications of Formal Verification

Functional Verification of Java Programs: Java Dynamic Logic

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2012

KIT – Institut für Theoretische Informatik



1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Loop Invariants
   - Basic Invariant Rule

Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program p
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of p:

- $[p]F$: If p terminates normally, then F holds in the final state ("partial correctness")
- $\langle p \rangle F$: p terminates normally, and F holds in the final state ("total correctness")

Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program p
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of p:

- $[p]F$: If p terminates normally, then F holds in the final state ("partial correctness")
- $\langle p \rangle F$: p terminates normally, and F holds in the final state ("total correctness")

Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program p
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of p:

- $[p]F$: If p terminates normally, then F holds in the final state ("partial correctness")
- $\langle p \rangle F$: p terminates normally, and F holds in the final state ("total correctness")

Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program p
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of p:

- $[p]F$: If p terminates normally, then F holds in the final state ("partial correctness")
- $\langle p \rangle F$: p terminates normally, and F holds in the final state ("total correctness")

Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

- Programs are “first-class citizens”
- Real Java syntax

Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Hoare triple $\{\psi\}\ \alpha\ \{\phi\}$ equiv. to DL formula ψ -> $[\alpha]\phi$

Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Not merely partial/total correctness:

- can employ programs for specification (e.g., verifying program transformations)
- can express security properties (two runs are indistinguishable)
- extension-friendly (e.g., temporal modalities)

Why Dynamic Logic?

Karlsruhe Institute of Technology

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Dynamic Logic Example Formulas

(`balance` >= c & `amount` > 0) ->
⟨`charge(amount);`⟩ `balance` > c

⟨`x = 1;`⟩([`while (true) {}`]*false*)

- Program formulas can appear nested

`\forall` *int val*; ((⟨`p`⟩`x` $\doteq$ *val*) <-> (⟨`q`⟩`x` $\doteq$ *val*))

- `p`, `q` equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

(`balance` $>=$ *c* & `amount` $>$ 0) $->$
$\langle$`charge(amount);`$\rangle$ `balance` $>$ *c*

$\langle$`x = 1;`$\rangle$([`while (true) {}`]*false*)

- Program formulas can appear nested

`\forall` *int val*; (($\langle$`p`$\rangle$`x` $\doteq$ *val*) $<->$ ($\langle$`q`$\rangle$`x` $\doteq$ *val*))

- `p`, `q` equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

(`balance` $>=$ c & `amount` $>$ 0) $->$
$\langle$`charge(amount);`$\rangle$ `balance` $>$ c

$\langle$`x = 1;`$\rangle$([`while (true) {}`]*false*)

- Program formulas can appear nested

`\forall` *int val*; (($\langle$`p`$\rangle$`x` $\doteq$ *val*) <-> ($\langle$`q`$\rangle$`x` $\doteq$ *val*))

- `p`, `q` equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \ \texttt{->}$
$\langle \texttt{charge(amount);} \rangle \ \texttt{balance} > c$

$\langle \texttt{x = 1;} \rangle ([\texttt{while (true) \{\}}]\mathit{false})$

- Program formulas can appear nested

$\texttt{\textbackslash forall}\ \mathit{int}\ \mathit{val}; ((\langle \texttt{p} \rangle \texttt{x} \doteq \mathit{val}) \ \texttt{<->} \ (\langle \texttt{q} \rangle \texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

(balance >= c & amount > 0) ->
⟨charge(amount);⟩ balance > c

⟨x = 1;⟩([while (true) {}]*false*)

- Program formulas can appear nested

\forall *int val*; ((⟨p⟩x $\doteq$ *val*) <-> (⟨q⟩x $\doteq$ *val*))

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

```
  a != null
->
  <
    int max = 0;
    if ( a.length > 0 ) max = a[0];
    int i = 1;
    while ( i < a.length ) {
      if ( a[i] > max ) max = a[i];
      ++i;
    }
  >(
      \forall int j; (j >= 0 & j < a.length -> max >= a[j])
      &
      (a.length > 0 ->
        \exists int j; (j >= 0 & j < a.length & max = a[j]))
    )
```

Variables

Karlsruhe Institute of Technology

- Logical variables disjoint from program variables
  - No quantification over program variables
  - Programs do not contain logical variables
  - “Program variables” actually non-rigid functions

Validity

A JAVA CARD DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas

Validity

A JAVA CARD DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas

Teil

Teil

Sequents and their Semantics

Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{\textit{Antecedent}} \quad \Longrightarrow \quad \underbrace{\phi_1, \ldots, \phi_n}_{\textit{Succedent}}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \;\&\; \cdots \;\&\; \psi_m) \quad \text{->} \quad (\phi_1 \;|\; \cdots \;|\; \phi_n)$$

Sequents and their Semantics

Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{\textit{Antecedent}} \quad \Longrightarrow \quad \underbrace{\phi_1, \ldots, \phi_n}_{\textit{Succedent}}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \;\; \& \;\; \cdots \;\; \& \;\; \psi_m) \quad \texttt{->} \quad (\phi_1 \;\; | \;\; \cdots \;\; | \;\; \phi_n)$$

Sequent Rules

General form

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion

Sequent Rules

General form

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion

Sequent Rules

General form

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion

Sequent Rules

General form

$$\text{rule\_name} \quad \overbrace{\frac{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}}^{\text{Premisses}}$$

($r = 0$ possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion

Some Simple Sequent Rules

$$\text{not\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\texttt{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal}\quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\quad \frac{\Gamma, \backslash\texttt{forall}\; t\; x; \phi,\; \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left}\quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\text{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal}\quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\quad \frac{\Gamma, \texttt{\textbackslash forall}\; t\; x; \phi,\; \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \texttt{\textbackslash forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

not_left $$\frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

imp_left $$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \text{ -> } B \Longrightarrow \Delta}$$

close_goal $$\frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

close_by_true $$\frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

all_left $$\frac{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\texttt{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left} \quad \frac{\Gamma, \backslash\texttt{forall}\; t\; x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

not_left $$\frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

imp_left $$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \texttt{->} B \Longrightarrow \Delta}$$

close_goal $$\frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

close_by_true $$\frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

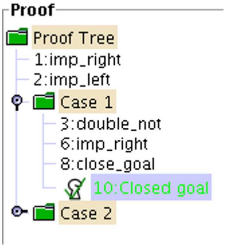
all_left $$\frac{\Gamma, \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Sequent Calculus Proofs

Proof tree

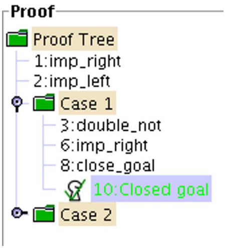
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Sequent Calculus Proofs

Proof tree

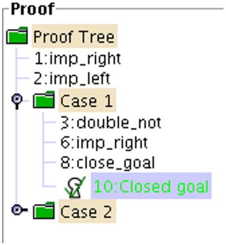
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Sequent Calculus Proofs

Proof tree

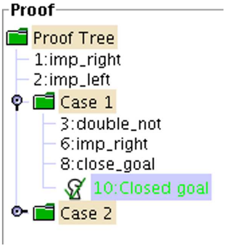
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Teil

Teil

Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

- Sequent rules execute symbolically the active statement

Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```
l:{try{ i=0; j=0; } finally{ k=0; }}
```

- Sequent rules execute symbolically the active statement

Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```
l:{try{ i=0; j=0; } finally{ k=0; }}
```

- Sequent rules execute symbolically the active statement

Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

passive prefix	π
active statement	`i=0;`
rest	ω

- Sequent rules execute symbolically the active statement

Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

passive prefix	π
active statement	`i=0;`
rest	ω

- Sequent rules execute symbolically the active statement

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = true \Longrightarrow \langle p \;\; \omega\rangle\phi, \Delta \qquad \Gamma, B = false \Longrightarrow \langle q \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{if (}B\texttt{) \{ } p \texttt{ \} else \{ } q \texttt{ \}} \;\; \omega\rangle\phi, \Delta}$$

Complicated statements/expressions are simplified first, e.g.

$$\frac{\Gamma \Longrightarrow \langle \texttt{v=y; y=y+1; x=v;} \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x=y++;} \;\; \omega\rangle\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\langle\omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle loc\texttt{=}val\texttt{;} \;\; \omega\rangle\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \mathit{true} \Longrightarrow \langle p \;\; \omega\rangle\phi, \Delta \qquad \Gamma, B = \mathit{false} \Longrightarrow \langle q \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{if (}B\texttt{) \{ } p \texttt{ \} else \{ } q \texttt{ \}} \;\; \omega\rangle\phi, \Delta}$$

Complicated statements/expressions are simplified first, e.g.

$$\frac{\Gamma \Longrightarrow \langle \texttt{v=y; y=y+1; x=v;} \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x=y++;} \;\; \omega\rangle\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\langle\omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle loc\texttt{=}val\texttt{;} \;\; \omega\rangle\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \mathit{true} \Longrightarrow \langle p \;\; \omega\rangle\phi, \Delta \qquad \Gamma, B = \mathit{false} \Longrightarrow \langle q \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{if (}B\texttt{) \{ } p \texttt{ \} else \{ } q \texttt{ \}} \;\; \omega\rangle\phi, \Delta}$$

Complicated statements/expressions are simplified first, e.g.

$$\frac{\Gamma \Longrightarrow \langle \texttt{v=y; y=y+1; x=v;} \;\; \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x=y++;} \;\; \omega\rangle\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{\mathit{loc} := \mathit{val}\}\langle\omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathit{loc}\texttt{=}\mathit{val}\texttt{;} \;\; \omega\rangle\phi, \Delta}$$

Treating Assignment with "Updates"

Updates

explicit syntactic elements in the logic

Elementary Updates

$$\{loc := val\}\,\phi$$

where (roughly)

- *loc* is a program variable *x*, an attribute access *o.attr*, or an array access *a*[*i*]
- *val* is same as *loc*, or a literal, or a logical variable

Parallel Updates

$$\{loc_1 := t_1 \mid\mid \cdots \mid\mid loc_n := t_n\}\,\phi$$

no dependency between the *n* components (but 'right wins' semantics)

Treating Assignment with "Updates"

Updates

explicit syntactic elements in the logic

Elementary Updates

$$\{loc := val\}\ \phi$$

where (roughly)

- *loc* is a program variable *x*, an attribute access *o.attr*, or an array access $a[i]$
- *val* is same as *loc*, or a literal, or a logical variable

Parallel Updates

$$\{loc_1 := t_1 \parallel \cdots \parallel loc_n := t_n\}\ \phi$$

no dependency between the *n* components (but 'right wins' semantics)

Treating Assignment with “Updates”

Updates

explicit syntactic elements in the logic

Elementary Updates

$$\{loc := val\}\,\phi$$

where (roughly)

- *loc* is a program variable *x*, an attribute access *o.attr*, or an array access $a[i]$
- *val* is same as *loc*, or a literal, or a logical variable

Parallel Updates

$$\{loc_1 := t_1 \mid\mid \cdots \mid\mid loc_n := t_n\}\,\phi$$

no dependency between the *n* components (but ‘right wins’ semantics)

Why Updates?

Updates are:

- *lazily applied* (i.e., substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages

- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)

Why Updates?

Updates are:

- *lazily applied* (i.e., substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages

- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)

Symbolic Execution with Updates
(by Example)

$$x < y \Longrightarrow x < y$$

$\vdots$

$$x < y \Longrightarrow \{x:=y \parallel y:=x\}\langle\rangle\, y < x$$

$\vdots$

$$x < y \Longrightarrow \{t:=x \parallel x:=y \parallel y:=x\}\langle\rangle\, y < x$$

$\vdots$

$$x < y \Longrightarrow \{t:=x \parallel x:=y\}\{y:=t\}\langle\rangle\, y < x$$

$\vdots$

$$x < y \Longrightarrow \{t:=x\}\{x:=y\}\langle y=t;\rangle\, y < x$$

$\vdots$

$$x < y \Longrightarrow \{t:=x\}\langle x=y;\ y=t;\rangle\, y < x$$

$\vdots$

$$\Longrightarrow x < y \texttt{->} \langle \texttt{int t=x; x=y; y=t;}\rangle\, y < x$$

Symbolic Execution with Updates
(by Example)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \ \mathtt{->}\ \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates
(by Example)

$$\texttt{x} < \texttt{y} \Longrightarrow \texttt{x} < \texttt{y}$$

⋮

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{x:=y} \,\|\, \texttt{y:=x}\}\langle\rangle\, \texttt{y} < \texttt{x}$$

⋮

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x} \,\|\, \texttt{x:=y} \,\|\, \texttt{y:=x}\}\langle\rangle\, \texttt{y} < \texttt{x}$$

⋮

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x} \,\|\, \texttt{x:=y}\}\{\texttt{y:=t}\}\langle\rangle\, \texttt{y} < \texttt{x}$$

⋮

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x}\}\{\texttt{x:=y}\}\langle\texttt{y=t;}\rangle\, \texttt{y} < \texttt{x}$$

⋮

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x}\}\langle\texttt{x=y; y=t;}\rangle\, \texttt{y} < \texttt{x}$$

⋮

$$\Longrightarrow \texttt{x} < \texttt{y} \;\texttt{->}\; \langle\texttt{int t=x; x=y; y=t;}\rangle\, \texttt{y} < \texttt{x}$$

Symbolic Execution with Updates
(by Example)

$$x < y \Longrightarrow x < y$$

$$\vdots$$

$$x < y \Longrightarrow \{x{:=}y \parallel y{:=}x\}\langle\rangle\, y < x$$

$$\vdots$$

$$x < y \Longrightarrow \{t{:=}x \parallel x{:=}y \parallel y{:=}x\}\langle\rangle\, y < x$$

$$\vdots$$

$$x < y \Longrightarrow \{t{:=}x \parallel x{:=}y\}\{y{:=}t\}\langle\rangle\, y < x$$

$$\vdots$$

$$x < y \Longrightarrow \{t{:=}x\}\{x{:=}y\}\langle y{=}t;\rangle\, y < x$$

$$\vdots$$

$$x < y \Longrightarrow \{t{:=}x\}\langle x{=}y;\ y{=}t;\rangle\, y < x$$

$$\vdots$$

$$\Longrightarrow x < y \;\texttt{->}\; \langle \texttt{int } t{=}x;\ x{=}y;\ y{=}t;\rangle\, y < x$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$\vdots$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \,||\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$\vdots$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \,||\, \mathtt{x:=y} \,||\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$\vdots$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \,||\, \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$\vdots$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$\vdots$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$\vdots$

$$\implies \mathtt{x} < \mathtt{y} \mathtt{->} \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates
(by Example)

$$\mathtt{x} < \mathtt{y} \implies \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \ \mathtt{->} \ \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$$

Symbolic Execution with Updates (by Example)

$$\mathtt{x} < \mathtt{y} \Longrightarrow \mathtt{x} < \mathtt{y}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \Longrightarrow \{\mathtt{x:=y} \,||\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \Longrightarrow \{\mathtt{t:=x} \,||\, \mathtt{x:=y} \,||\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \Longrightarrow \{\mathtt{t:=x} \,||\, \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \Longrightarrow \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \Longrightarrow \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\Longrightarrow \mathtt{x} < \mathtt{y} \,\texttt{->}\, \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}}\ \ \omega\end{array}\right\rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\rangle \phi, \Delta}$$

Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}}\ \ \omega\end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\rangle \phi, \Delta}$$

Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\pi\ \texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}}\quad \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega \rangle \phi, \Delta}$$

Teil

1. JAVA CARD DL
2. Sequent Calculus
3. **Rules for Programs: Symbolic Execution**
4. A Calculus for 100% JAVA CARD
5. Loop Invariants
   - Basic Invariant Rule

Teil

Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY

Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY

Java—A Language of Many Features

Karlsruhe Institute of Technology

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: Feature needs not be handled in calculus
Contra: Modified source code
Example in KeY: Very rare: treating inner classes

Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: Flexible, easy to implement, usable
Contra: Not expressive enough for all features
Example in KeY: Complex expression eval, method inlining, etc., etc.

Java—A Language of Many Features

Karlsruhe Institute of Technology

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: No logic extensions required, enough to express most features
Contra: Creates difficult first-order POs, unreadable antecedents
Example in KeY: Dynamic types and branch predicates

Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: Arbitrarily expressive extensions possible
Contra: Increases complexity of all rules
Example in KeY: Method frames, updates

Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - using loop invariants
   - using induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification

Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - using loop invariants
   - using induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification

Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - using loop invariants
   - using induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification

Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - using loop invariants
   - using induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification

Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules
2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates
3. Rules for handling loops
   - using loop invariants
   - using induction
4. Rules for replacing a method invocations by the method's contract
5. Update simplification

Loop Invariants

Symbolic execution of loops: unwind

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

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

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? 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

Symbolic execution of loops: unwind

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

How to handle a loop with…

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

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? 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

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

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? 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

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?

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

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 $\times$
- 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

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? 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*

Basic Invariant Rule

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

Loop Invariants Cont'd

Karlsruhe Institute of Technology

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

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

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

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

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

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

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

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

Loop Invariants Cont'd

Karlsruhe Institute of Technology

Basic Invariant Rule: Problem

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

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise
- *But:* context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

Loop Invariants Cont'd

Basic Invariant Rule: Problem

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

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise
- *But:* context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

Loop Invariants Cont'd

Karlsruhe Institute of Technology

Basic Invariant Rule: Problem

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

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise
- *But:* context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

Loop Invariants Cont'd

Basic Invariant Rule: Problem

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

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

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

Example

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

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

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

Example

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

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

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

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

Example

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

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

```
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: $\texttt{a} \doteq\!\!\!\!\!/\; \textbf{null}$ & *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: $0 \leq \texttt{i}$ & $\texttt{i} \leq \texttt{a.length}$
& $\forall\, \textbf{int}\ x;\ (0 \leq x < \texttt{i} \rightarrow \texttt{a}[x] \doteq 1)$
& $\texttt{a} \doteq\!\!\!\!\!/\; \textbf{null}$
& *ClassInv'*

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[*];
```

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[*];
```

Example with Improved Invariant Rule

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

Example with Improved Invariant Rule

Precondition: $a \doteq\not$ **`null`**

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

Example with Improved Invariant Rule

Precondition: $\mathtt{a} \mathrel{\dot{\neq}} \mathbf{null}$

```
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 with Improved Invariant Rule

Precondition: $\mathtt{a} \not\doteq \mathbf{null}$

```
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: $\mathtt{a} \neq \mathtt{null}$

```
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: $\texttt{a} \doteq\!\!\!\!/\ \textbf{null}\ \&\ \textit{ClassInv}$

```
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

$\forall$ `int` x;
  $($`n` $\doteq x \wedge x >= 0 \rightarrow$
    [`i = 0; r = 0;`

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

   ]`r` $\doteq$?)

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`

Example

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

```
[ 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`

Example

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

```
[ 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`

Example

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

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;`

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 premises 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
```

Total Correctness

Find a decreasing integer term v (called *variant*)

Add the following premises 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 premises 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`