Applications of Formal Verification

Functional Verification of Java Programs: Java Dynamic Logic

Bernhard Beckert · Mattias Ulbrich | SS 2019
1. **Java Card DL**

2. **Sequent Calculus**

3. **Rules for Programs: Symbolic Execution**

4. **A Calculus for 100% Java Card**

5. **Loop Invariants**
1. **Java Card DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% **Java Card**

5. Loop Invariants
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

- 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

- 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 \( \psi \rightarrow [\alpha]\phi \)
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?

- 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

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount)} \rangle \text{balance} > c\]

\[\langle x = 1;\rangle ([\text{while (true) {}}] \text{false})\]
- Program formulas can appear nested

\[\forall \text{int } val; (\langle p \rangle x = val) \iff (\langle q \rangle x = val)\]
- \(p, q\) equivalent relative to computation state restricted to \(x\)
Dynamic Logic Example Formulas

\[(balance \geq c \land amount > 0) \rightarrow \langle \text{charge(amount);} \rangle balance > c\]

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

- Program formulas can appear nested

\[\forall int val; ((\langle p \rangle x = val) \leftrightarrow (\langle q \rangle x = val))\]

- \(p, q\) equivalent relative to computation state restricted to \(x\)
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(\text{amount});} \rangle \text{balance} > c\]

\[\langle x = 1; \rangle ([\text{while (true) \{\}\}]) \text{false}\]
- Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x = \text{val}) \leftrightarrow (\langle q \rangle x = \text{val}))\]
- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount)} ; \rangle \text{balance} > c\]

\[\langle x = 1 ; \rangle ([\text{while (true) {} }] \text{true})\]
- Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x = \text{val}) \leftrightarrow (\langle q \rangle x = \text{val}))\]
- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount)}; \rangle \text{balance} > c\]

\[\langle x = 1; \rangle (\text{[while (true) {}]} \langle false \rangle)\]
- Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x = \text{val}) \iff (\langle q \rangle x = \text{val}))\]
- \(p, q\) equivalent relative to computation state restricted to \(x\)
Dynamic Logic Example Formulas

\( a \neq \text{null} \)

\[ \implies \]

\[
\begin{align*}
\text{int} & \quad \text{max} = 0; \\
\text{if} & \quad (a\text{.length} > 0) \quad \text{max} = a[0]; \\
\text{int} & \quad i = 1; \\
\text{while} & \quad (i < a\text{.length}) \{
\quad \text{if} & \quad (a[i] > \text{max}) \quad \text{max} = a[i]; \\
\quad & \quad ++i;
\}
\end{align*}
\]

\[ \implies \]

\[
\forall \text{int } j; (j \geq 0 \& j < a\text{.length} \implies \text{max} \geq a[j]) \\&
\]

\[
(a\text{.length} > 0 \implies \\
\exists \text{int } j; (j \geq 0 \& j < a\text{.length} \& \text{max} = a[j]))
\]
Variables

- Logical variables disjoint from program variables
  - No quantification over program variables
  - Programs do not contain logical variables
  - “Program variables” actually non-rigid functions
A Java CARD DL formula is valid iff it is true in all states.
A Java Card DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
1. **JAVA CARD DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% JAVA CARD

5. Loop Invariants
1. **JAVA CARD DL**

2. **Sequent Calculus**

3. **Rules for Programs: Symbolic Execution**

4. **A Calculus for 100% JAVA CARD**

5. **Loop Invariants**
Sequents and their Semantics

Syntax

\[ \psi_1, \ldots, \psi_m \quad \Rightarrow \quad \phi_1, \ldots, \phi_n \]

where the \( \phi_i, \psi_i \) are formulae (without free variables)

Semantics

Same as the formula

\[ (\psi_1 \land \cdots \land \psi_m) \quad \Rightarrow \quad (\phi_1 \lor \cdots \lor \phi_n) \]
Sequents and their Semantics

Syntax

\[ \psi_1, \ldots, \psi_m \Rightarrow \phi_1, \ldots, \phi_n \]

where the \( \phi_i, \psi_i \) are formulae (without free variables)

Semantics

Same as the formula

\[ (\psi_1 \land \cdots \land \psi_m) \Rightarrow (\phi_1 \lor \cdots \lor \phi_n) \]
Sequent Rules

General form

RULE_NAME

Γ₁ ⇒ Δ₁  \cdots  Γᵣ ⇒ Δᵣ

Γ ⇒ Δ

Premisses

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**

\[
\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r}{\Gamma \Rightarrow \Delta}
\]

* RULE_NAME *

Premisses

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

\[ \text{RULE NAME} \]
\[ \begin{array}{c}
\Gamma_1 \Rightarrow \Delta_1 \\
\cdots \\
\Gamma_r \Rightarrow \Delta_r \\
\hline
\Gamma \Rightarrow \Delta
\end{array} \]

(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

RULE_NAME

Premisses

Γ₁ ⇒ Δ₁  ···  Γᵣ ⇒ Δᵣ

Γ ⇒ Δ

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
Some Simple Sequent Rules

**NOT_LEFT**

\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

**IMP_LEFT**

\[
\frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \to B \Rightarrow \Delta}
\]

**CLOSE_GOAL**

\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

**CLOSE_BY_TRUE**

\[
\frac{\Gamma \Rightarrow \text{true, } \Delta}{\Gamma \Rightarrow \text{true, } \Delta}
\]

**ALL_LEFT**

\[
\frac{\Gamma, \forall t \ x ; \phi , \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \ x ; \phi \Rightarrow \Delta}
\]

where \( e \) is var-free term of type \( t' \prec t \)
Some Simple Sequent Rules

\[
\text{NOT\_LEFT} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

\[
\text{IMP\_LEFT} \quad \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

\[
\text{CLOSE\_GOAL} \quad \frac{}{\Gamma, A \Rightarrow A, \Delta}
\]

\[
\text{CLOSE\_BY\_TRUE} \quad \frac{}{\Gamma \Rightarrow \text{true}, \Delta}
\]

\[
\text{ALL\_LEFT} \quad \frac{\Gamma, \forall t \ x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \ x; \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' < t\)
Some Simple Sequent Rules

**NOT_LEFT**

\[
\Gamma \Rightarrow A, \Delta \\
\therefore \Gamma, \neg A \Rightarrow \Delta
\]

**IMP_LEFT**

\[
\Gamma \Rightarrow A, \Delta, \Gamma, B \Rightarrow \Delta \\
\therefore \Gamma, A \rightarrow B \Rightarrow \Delta
\]

**CLOSE_GOAL**

\[
\Gamma, A \Rightarrow A, \Delta \\
\therefore \Gamma \Rightarrow true, \Delta
\]

**ALL_LEFT**

\[
\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta \\
\therefore \Gamma, \forall t x; \phi \Rightarrow \Delta
\]

where \( e \) var-free term of type \( t' \prec t \)
Some Simple Sequent Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOT_LEFT</strong></td>
<td>[ \Gamma \Rightarrow A, \Delta ]</td>
<td>[ \Gamma, \neg A \Rightarrow \Delta ]</td>
</tr>
<tr>
<td><strong>IMP_LEFT</strong></td>
<td>[ \Gamma \Rightarrow A, \Delta ], [ \Gamma, B \Rightarrow \Delta ]</td>
<td>[ \Gamma, A \rightarrow B \Rightarrow \Delta ]</td>
</tr>
<tr>
<td><strong>CLOSE_GOAL</strong></td>
<td>[ \Gamma, A \Rightarrow A, \Delta ]</td>
<td></td>
</tr>
<tr>
<td><strong>CLOSE_BY_TRUE</strong></td>
<td>[ \Gamma \Rightarrow \text{true}, \Delta ]</td>
<td></td>
</tr>
<tr>
<td><strong>ALL_LEFT</strong></td>
<td>[ \Gamma, \forall t x; \phi, {x/e} \phi \Rightarrow \Delta ]</td>
<td>[ \Gamma, \forall t x; \phi \Rightarrow \Delta ]</td>
</tr>
</tbody>
</table>

where \( e \) var-free term of type \( t' \prec t \)
Some Simple Sequent Rules

\[
\text{NOT\_LEFT} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

\[
\text{IMP\_LEFT} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta} \quad \frac{\Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

\[
\text{CLOSE\_GOAL} \quad \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

\[
\text{CLOSE\_BY\_TRUE} \quad \frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

\[
\text{ALL\_LEFT} \quad \frac{\Gamma, \forall t \ x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \ x; \phi \Rightarrow \Delta}
\]

where \( e \) var-free term of type \( t' < t \)
**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
1. **JAVA CARD DL**

2. **Sequent Calculus**

3. **Rules for Programs: Symbolic Execution**

4. **A Calculus for 100% JAVA CARD**

5. **Loop Invariants**
1. **JAVA CARD DL**

2. Sequent Calculus

3. **Rules for Programs: Symbolic Execution**

4. **A Calculus for 100% JAVA CARD**

5. Loop Invariants
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

\[
1:\text{try}\{ \ i=0; \ j=0; \ \} \ \text{finally}\{ \ k=0; \ \}
\]

- 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

\[ l: \{ \text{try} \{ \ i=0; \ j=0; \ \} \ \text{finally} \{ \ k=0; \ \} \} \]

- 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

\[
\begin{align*}
&\text{passive prefix } \pi \\
&\text{active statement } i=0; \\
&\text{rest } \omega \\
\end{align*}
\]

Sequential 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

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

- Passive prefix: $\pi$
- Active statement: $i=0$
- Rest: $\omega$

- Sequent rules execute symbolically the active statement
Rules for Symbolic Program Execution

If-then-else rule

\[ \Gamma, B = \text{true} \implies \langle p \; \omega \rangle \phi, \Delta \quad \Gamma, B = \text{false} \implies \langle q \; \omega \rangle \phi, \Delta \]

\[ \Gamma \implies \langle \text{if (B) } \{ \; p \; \} \text{ else } \{ \; q \; \} \; \omega \rangle \phi, \Delta \]

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

\[ \Gamma \implies \langle v=y; \; y=y+1; \; x=v; \; \omega \rangle \phi, \Delta \]

\[ \Gamma \implies \langle x=y++; \; \omega \rangle \phi, \Delta \]

Simple assignment rule

\[ \Gamma \implies \{ \text{loc := val} \} \langle \omega \rangle \phi, \Delta \]

\[ \Gamma \implies \langle \text{loc=val}; \; \omega \rangle \phi, \Delta \]
# Rules for Symbolic Program Execution

## If-then-else rule

\[
\Gamma, B = \text{true} \implies \langle p \; \omega \rangle \phi, \Delta \\
\Gamma, B = \text{false} \implies \langle q \; \omega \rangle \phi, \Delta \\
\Gamma \implies \langle \text{if (B) } \{ p \} \text{ else } \{ q \} \; \omega \rangle \phi, \Delta
\]

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

\[
\Gamma \implies \langle v=y; \; y=y+1; \; x=v; \; \omega \rangle \phi, \Delta \\
\Gamma \implies \langle x=y++; \; \omega \rangle \phi, \Delta
\]

## Simple assignment rule

\[
\Gamma \implies \{ loc := val \} \langle \omega \rangle \phi, \Delta \\
\Gamma \implies \langle loc=\text{val}; \; \omega \rangle \phi, \Delta
\]
Rules for Symbolic Program Execution

If-then-else rule

\[
\frac{\Gamma, B = true \Rightarrow \langle p \ \omega \rangle \phi, \Delta \quad \Gamma, B = false \Rightarrow \langle q \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \text{if} \ (B) \ \{ p \} \ \text{else} \ \{ q \} \ \omega \rangle \phi, \Delta}
\]

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

\[
\Gamma \Rightarrow \langle v=y; \ y=y+1; \ x=v; \ \omega \rangle \phi, \Delta
\]

\[
\frac{\Gamma \Rightarrow \langle x=y++; \ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle x=y++; \ \omega \rangle \phi, \Delta}
\]

Simple assignment rule

\[
\frac{\Gamma \Rightarrow \{ loc := val \} \langle \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle loc=val; \ \omega \rangle \phi, \Delta}
\]
Updates

syntactic elements in the logic – (explicit substitutions)

Elementary Updates

\{ loc := val \} \phi

where

- \textit{loc} is a program variable
- \textit{val} is an expression type-compatible with \textit{loc}

Parallel Updates

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

no dependency between the \textit{n} components (but ‘last wins’ semantics)
Treating Assignment with “Updates”

Updates
syntactic elements in the logic – (explicit substitutions)

Elementary Updates

\[
\{ \text{loc} := \text{val} \} \phi
\]

where
- \text{loc} is a program variable
- \text{val} is an expression type-compatible with \text{loc}

Parallel Updates

\[
\{ \text{loc}_1 := t_1 \parallel \cdots \parallel \text{loc}_n := t_n \} \phi
\]

no dependency between the \( n \) components (but ‘last wins’ semantics)
Treating Assignment with “Updates”

Updates
syntactic elements in the logic – (explicit substitutions)

Elementary Updates

\[ \{ \text{loc} := \text{val} \} \phi \]

where
- \text{loc} is a program variable
- \text{val} is an expression type-compatible with \text{loc}

Parallel Updates

\[ \{ \text{loc}_1 := t_1 \ \| \ \cdots \ \| \ \text{loc}_n := t_n \} \phi \]

no dependency between the \( n \) components (but ‘last wins’ semantics)
Why Updates?

Updates are

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

Advantages

- no renaming required
  (compared to another forward proof technique: strongest-postcondition calculus)
- delayed/minimised proof branching
  efficient aliasing treatment)
Why Updates?

Updates are

- aggregations of state change
- eagerly parallelised + simplified
- lazily applied (i.e., substituted into postcondition)

Advantages

- no renaming required
  (compared to another forward proof technique: strongest-postcondition calculus)
- delayed/minimised proof branching
efficient aliasing treatment)
Symbolic Execution with Updates
(by Example)

\[
x < y \implies x < y
\]

\[
x < y \implies \{x := y \mid y := x\}\langle \rangle y < x
\]

\[
x < y \implies \{t := x \mid x := y \mid y := x\}\langle \rangle y < x
\]

\[
x < y \implies \{t := x \mid x := y\}\{y := t\}\langle \rangle y < x
\]

\[
x < y \implies \{t := x\}\{x := y\}\langle y = t; \rangle y < x
\]

\[
x < y \implies \{t := x\}\langle x = y; \ y = t; \rangle y < x
\]

\[
\implies x < y \implies \langle \text{int } t=x; \ x=y; \ y=t; \rangle y < x
\]
Symbolic Execution with Updates (by Example)

\[ x < y \implies x < y \]

\[ \vdots \]

\[ x < y \implies \{ x := y \parallel y := x \} \langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \parallel x := y \parallel y := x \} \langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \parallel x := y \} \{ y := t \} \langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \} \{ x := y \} \langle y = t ; \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \} \langle x = y ; y = t ; \rangle y < x \]

\[ \implies x < y \rightarrow \langle \text{int} \ t = x ; x = y ; y = t ; \rangle y < x \]
Symbolic Execution with Updates (by Example)

\[
x < y \Rightarrow x < y
\]

\[
\vdots
\]

\[
x < y \Rightarrow \{x := y \parallel y := x\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \Rightarrow \{t := x \parallel x := y \parallel y := x\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \Rightarrow \{t := x \parallel x := y\} \{y := t\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \Rightarrow \{t := x\} \{x := y\} \langle y = t; \rangle y < x
\]

\[
\vdots
\]

\[
x < y \Rightarrow \{t := x\} \langle x = y; \ y = t; \rangle y < x
\]

\[
\vdots
\]

\[
\Rightarrow x < y \rightarrow \langle \text{int } t = x; \ x = y; \ y = t; \rangle y < x
\]
Symbolic Execution with Updates
(by Example)

\[
x < y \implies x < y
\]

\[
\vdots
\]

\[
x < y \implies \{x := y \parallel y := x\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x \parallel x := y \parallel y := x\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x \parallel x := y\}\{y := t\} \langle \rangle y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x\}\{x := y\}\{y := t\}\langle y = t; \rangle y < x
\]

\[
\vdots
\]

\[
x < y \implies \{t := x\}\langle x = y; y = t; \rangle y < x
\]

\[
\vdots
\]

\[
\Rightarrow x < y \rightarrow \langle \text{int } t = x; x = y; y = t; \rangle y < x
\]
Symbolic Execution with Updates (by Example)

\[
x < y \implies x < y
\]
\[
\vdots
\]
\[
x < y \implies \{ x := y \mid y := x \} \langle \rangle \ y < x
\]
\[
\vdots
\]
\[
x < y \implies \{ t := x \mid x := y \mid y := x \} \langle \rangle \ y < x
\]
\[
\vdots
\]
\[
x < y \implies \{ t := x \mid y := t \} \langle \rangle \ y < x
\]
\[
\vdots
\]
\[
x < y \implies \{ t := x \} \{ x := y \} \langle y = t ; \rangle \ y < x
\]
\[
\vdots
\]
\[
x < y \implies \{ t := x \} \langle x = y ; \ y = t ; \rangle \ y < x
\]
\[
\vdots
\]
\[
\Rightarrow x < y \to \langle \text{int} \ t = x ; \ x = y ; \ y = t ; \rangle \ y < x
\]
Symbolic Execution with Updates (by Example)

\[ x < y \implies x < y \]

\[ x < y \implies \{ x := y \parallel y := x \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \parallel x := y \parallel y := x \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \parallel x := y \} \{ y := t \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \} \{ x := y \} \langle y = t ; \rangle \ y < x \]

\[ x < y \implies \{ t := x \} \langle x = y ; \ y = t ; \rangle \ y < x \]

\[ \implies x < y \rightarrow \langle \text{int} \ t = x ; \ x = y ; \ y = t ; \rangle \ y < x \]
Symbolic Execution with Updates
(by Example)

\[ x < y \implies x < y \]

\[ \vdots \]

\[ x < y \implies \{x:=y \parallel y:=x\}\langle \rangle \ y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x \parallel x:=y \parallel y:=x\}\langle \rangle \ y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x \parallel x:=y\}\{y:=t\}\langle \rangle \ y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x\}\{x:=y\}\langle y=t; \rangle \ y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x\}\langle x=y; \ y=t; \rangle \ y < x \]

\[ \implies x < y \rightarrow \langle \text{int} \ t=x; \ x=y; \ y=t; \rangle \ y < x \]
The theory of arrays

An abstract datatype \( \text{Array}(\mathbb{I}, \mathbb{V}) \)

**Types:** Indices \( \mathbb{I} \), Values \( \mathbb{V} \)

**Function symbols:**
- \( \text{select} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \rightarrow \mathbb{V} \)
- \( \text{store} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \times \mathbb{V} \rightarrow \text{Array}(\mathbb{I}, \mathbb{V}) \)

**Axioms**

\[
\forall a, i, v. \quad \text{select}(\text{store}(a, i, v), i) = v
\]

\[
\forall a, i, j, v. \quad i \neq j \rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)
\]

**Intuition**

\( \mathcal{D}(\text{Array}(\mathbb{I}, \mathbb{V})) \) represents the set of functions \( \mathcal{D}(\mathbb{I}) \rightarrow \mathcal{D}(\mathbb{V}) \)
The theory of arrays

An abstract datatype $Array(\mathbb{I}, \mathbb{V})$

Types: Indices $\mathbb{I}$, Values $\mathbb{V}$

Function symbols:
- $select : Array(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \rightarrow \mathbb{V}$
- $store : Array(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \times \mathbb{V} \rightarrow Array(\mathbb{I}, \mathbb{V})$

Axioms

\[ \forall a, i, v. \quad select(store(a, i, v), i) = v \]
\[ \forall a, i, j, v. \ i \neq j \rightarrow select(store(a, i, v), j) = select(a, j) \]

Intuition

$D(Array(\mathbb{I}, \mathbb{V}))$ represents the set of functions $D(\mathbb{I}) \rightarrow D(\mathbb{V})$
The theory of arrays

An abstract datatype $\text{Array}(\mathbb{I}, \mathbb{V})$

**Types:** Indices $\mathbb{I}$, Values $\mathbb{V}$

**Function symbols:**
- $\text{select} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \to \mathbb{V}$
- $\text{store} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \times \mathbb{V} \to \text{Array}(\mathbb{I}, \mathbb{V})$

**Axioms**

$\forall a, i, v. \quad \text{select}(\text{store}(a, i, v), i) = v$

$\forall a, i, j, v. \ i \neq j \to \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)$

**Intuition**

$\mathcal{D}(\text{Array}(\mathbb{I}, \mathbb{V}))$ represents the set of functions $\mathcal{D}(\mathbb{I}) \to \mathcal{D}(\mathbb{V})$
The theory of arrays

An abstract datatype

Types: Indices $\mathbb{I}$, Values $\mathbb{V}$

Function symbols:
- $\text{select} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \rightarrow \mathbb{V}$
- $\text{store} : \text{Array}(\mathbb{I}, \mathbb{V}) \times \mathbb{I} \times \mathbb{V} \rightarrow \text{Array}(\mathbb{I}, \mathbb{V})$

Axioms

$\forall a, i, v. \quad \text{select}(\text{store}(a, i, v), i) = v$

$\forall a, i, j, v. \ i \neq j \rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)$

Intuition

$\mathcal{D}(\text{Array}(\mathbb{I}, \mathbb{V}))$ represents the set of functions $\mathcal{D}(\mathbb{I}) \rightarrow \mathcal{D}(\mathbb{V})$

John McCarthy (1927–2011): Theory of arrays is decidable
Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays: $\Pi = \text{Object} \times \text{Field}$, $\forall = \text{Any}$

$\text{heap} : \text{Heap}$ (the heap in the current state)

$\text{select} : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any}$

$\text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap}$

Some special program variables

$\text{self}$ the current receiver object (this in Java)

$\text{exc}$ the currently active exception (null if none thrown)

$\text{result}$ the result of the method invocation
### Program State Representation

#### Local program variables
Modeled as non-rigid constants

#### Heap
Modeled with theory of arrays: $\mathbb{I} = Object \times Field$, $\mathbb{V} = Any$

- **heap**: $Heap$ (the heap in the current state)
- **select**: $Heap \times Object \times Field \rightarrow Any$
- **store**: $Heap \times Object \times Field \times Any \rightarrow Heap$

#### Some special program variables
- **self**: the current receiver object (**this** in Java)
- **exc**: the currently active exception (**null** if none thrown)
- **result**: the result of the method invocation
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays: \( \mathbb{I} = \text{Object} \times \text{Field}, \mathbb{V} = \text{Any} \)

- **heap**: Heap (the heap in the current state)
- **select**: Heap \( \times \) Object \( \times \) Field \( \rightarrow \) Any
- **store**: Heap \( \times \) Object \( \times \) Field \( \times \) Any \( \rightarrow \) Heap

Some special program variables

- **self**: the current receiver object (*this in Java*)
- **exc**: the currently active exception (*null if none thrown*)
- **result**: the result of the method invocation
1. **Java Card DL**

2. **Sequent Calculus**

3. **Rules for Programs: Symbolic Execution**

4. **A Calculus for 100% Java Card**

5. **Loop Invariants**
1. **JAVA CARD DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. **A Calculus for 100% JAVA CARD**

5. Loop Invariants
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

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

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**
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

<table>
<thead>
<tr>
<th>UNWIND_LOOP</th>
<th>$\Gamma \Rightarrow U[\pi \text{if}(b) \ { p; \ while(b) \ p} \ \omega]\phi, \Delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Gamma \Rightarrow U[\pi \text{while}(b) \ p \ \omega]\phi, \Delta$</td>
</tr>
</tbody>
</table>

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

\[ \Gamma \Rightarrow U[\pi \text{ if } (b) \{ p; \text{ while } (b) \ p \} \omega ] \phi, \Delta \]

\[ \Gamma \Rightarrow U[\pi \text{ while } (b) \ p \omega ] \phi, \Delta \]

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

\[
\text{UNWIND LOOP} \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta
\]

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

\[
\text{UNWINDLOOP} \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta
\]

\[
\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\]

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

\[
\text{UNWIND LOOP} \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta
\]

\[
\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\]

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

\[
\text{UNWINDLOOP} \quad \frac{\Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta}{\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta}
\]

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

\[
\text{UNWINDLOOP} \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta
\]

\[
\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\]

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

\[
\begin{align*}
\text{UNWINDLOOP} & \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta \\
\text{UNWIND} & \quad \Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\end{align*}
\]

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

\[
\text{UNWIND LOOP} \quad \Gamma \Rightarrow U[\pi \text{if}(b) \{ p; \text{while}(b) p \} \omega] \phi, \Delta \\
\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\]

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 \( Inv \) that
  - holds initially and
  - whose validity is preserved by loop iteration
- 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
- Make \( Inv \) strong enough to entail the desired postcondition

Basic Invariant Rule

\[
\begin{align*}
\Gamma &\Rightarrow U \text{Inv}, \Delta \quad \text{(initially valid)} \\
\text{Inv, } b = \text{TRUE} &\Rightarrow [p] \text{Inv} \quad \text{(preserved)} \\
\text{Inv, } b = \text{FALSE} &\Rightarrow [\pi \omega] \phi \quad \text{(use case)}
\end{align*}
\]

\[\text{loopInvariant} \frac{\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta}{\Gamma \Rightarrow U \text{Inv}, \Delta} \]
Loop Invariants Cont’d

Idea behind loop invariants

- A formula $\textit{Inv}$ that
  - holds initially and
  - whose validity is preserved by loop iteration
- **Consequence:** if $\textit{Inv}$ was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then $\textit{Inv}$ holds afterwards
- Make $\textit{Inv}$ strong enough to entail the desired postcondition

Basic Invariant Rule

\[
\begin{align*}
\Gamma \Rightarrow U \textit{Inv}, \Delta \\
\textit{inv}, \ b \doteq \text{TRUE} \Rightarrow [p] \textit{Inv} \\
\textit{inv}, \ b \doteq \text{FALSE} \Rightarrow [\pi \omega] \phi \\
\text{loopInv} \Rightarrow \Gamma \Rightarrow U[\pi \text{while}(b) \ p \ \omega] \phi, \Delta
\end{align*}
\]
Loop Invariants Cont’d

Idea behind loop invariants

- A formula $Inv$ that
  - holds initially and
  - whose validity is preserved by loop iteration

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

- Make $Inv$ strong enough to entail the desired postcondition

Basic Invariant Rule

\[
\Gamma \implies U \begin{aligned}
&Inv, \Delta \\
&\text{(initially valid)}
\end{aligned}
\]

\[
\begin{align*}
&Inv, b \doteq \text{TRUE} \implies [p]Inv \\
&\text{(preserved)} \\
&\text{loopInvariant} \\
&Inv, b \doteq \text{FALSE} \implies [\pi \omega]\phi \\
&\text{(use case)} \\
&\Gamma \implies U[\pi \text{while}(b) \ p \ \omega]\phi, \Delta
\end{align*}
\]
Loop Invariants Cont’d

Idea behind loop invariants

- A formula $\text{Inv}$ that
  - holds initially and
  - whose validity is preserved by loop iteration
- **Consequence**: if $\text{Inv}$ was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then $\text{Inv}$ holds *afterwards*
- Make $\text{Inv}$ strong enough to entail the desired *postcondition*

<table>
<thead>
<tr>
<th>Basic Invariant Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \Rightarrow \mathcal{U} \text{Inv}, \Delta$ (initially valid)</td>
</tr>
<tr>
<td>$\text{Inv}, b \doteq \text{TRUE} \Rightarrow [p] \text{Inv}$ (preserved)</td>
</tr>
<tr>
<td>$\text{loopInvariant}$</td>
</tr>
<tr>
<td>$\text{Inv}, b \doteq \text{FALSE} \Rightarrow [\pi \omega] \phi$ (use case)</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow \mathcal{U} [\pi \text{while}(b) \ p \omega] \phi, \Delta$</td>
</tr>
</tbody>
</table>
Loop Invariants Cont’d

Idea behind loop invariants

- A formula $Inv$ that
  - holds initially and
  - whose validity is *preserved* by loop iteration
- 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*
- Make $Inv$ strong enough to entail the desired *postcondition*

Basic Invariant Rule

\[
\begin{align*}
\Gamma & \Rightarrow U \ Inv, \ \Delta \quad \text{(initially valid)} \\
Inv, \ b & \vdash TRUE \Rightarrow [p] Inv \quad \text{(preserved)} \\
Inv, \ b & \vdash FALSE \Rightarrow [\pi \omega] \phi \quad \text{(use case)} \\
\text{loopInvariant} \quad & \Gamma \Rightarrow U[\pi \ while (b) \ p \ \omega] \phi, \ \Delta
\end{align*}
\]
Basic Invariant Rule: Problem

\[ \begin{align*}
\Gamma &\implies \mathcal{U} \text{Inv}, \Delta \\
\text{Inv}, \ b \vdash \text{TRUE} &\implies [p]\text{Inv} \\
\text{Inv}, \ b \vdash \text{FALSE} &\implies [\pi \omega]\phi \\
\Gamma &\implies \mathcal{U} [\pi \text{while}(b) \ p \omega]\phi, \Delta
\end{align*} \]

- Context \( \Gamma, \Delta, \mathcal{U} \) must be omitted in 2nd and 3rd premise
- \textit{But}: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \textit{Inv}
Basic Invariant Rule: Problem

\[
\begin{align*}
\Gamma & \Rightarrow U \text{Inv}, \Delta \\
\text{Inv}, \; b = \text{TRUE} & \Rightarrow [p]\text{Inv} \\
\text{Inv}, \; b = \text{FALSE} & \Rightarrow [\pi \; \omega]\phi \\
\Gamma & \Rightarrow U[\pi \text{while}(b) \; p \; \omega]\phi, \Delta
\end{align*}
\]

- Context $\Gamma$, $\Delta$, $U$ 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 $\text{Inv}$
Loop Invariants Cont’d

Basic Invariant Rule: Problem

\[ \Gamma \Rightarrow U \text{inv}, \Delta \]  
\[ \text{inv}, b \models \text{TRUE} \Rightarrow [p \text{inv}] \]  
\[ \text{inv}, b \models \text{FALSE} \Rightarrow [\pi \text{while}(b) \ p \ \omega] \phi, \Delta \]  

- Context \( \Gamma, \Delta, U \) must be omitted in 2nd and 3rd premise
- \textit{But}: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \text{inv}
Loop Invariants Cont’d

Basic Invariant Rule: Problem

\[ \Gamma \implies \mathcal{U} \text{Inv}, \Delta \quad \text{(initially valid)} \]
\[ \text{Inv, } b \doteq \text{TRUE} \implies \lbrack p \rbrack \text{Inv} \quad \text{(preserved)} \]
\[ \text{Inv, } b \doteq \text{FALSE} \implies \lbrack \pi \omega \rbrack \phi \quad \text{(use case)} \]
\[ \Gamma \implies \mathcal{U} \lbrack \pi \text{while}(b) \ p \omega \rbrack \phi, \Delta \]

- Context \( \Gamma, \Delta, \mathcal{U} \) 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 \( \text{Inv} \)
Example

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

Precondition: \( a \neq \text{null} \)

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

Precondition: \( a \neq \text{null} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a.\text{length} \rightarrow a[x] = 1) \)
**Example**

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

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

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

**Loop invariant:** $0 \leq i \land i \leq a.length$
Example

Precondition: \( a \neq \text{null} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a.\text{length} \rightarrow a[x] = 1) \)

Loop invariant: \( 0 \leq i \wedge i \leq a.\text{length} \wedge \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
Example

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

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

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

Loop invariant: $0 \leq i \land i \leq a\text{.length}$

$\land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1)$

$\land a \neq \text{null}$
Example

Precondition: \( a \neq \text{null} \) & ClassInv

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a\text{.length} \rightarrow a[x] = 1) \)

Loop invariant: \( 0 \leq i \land i \leq a\text{.length} \land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \land a \neq \text{null} \land \text{ClassInv}' \)
Want to keep part of the context that is *unmodified* by loop

*assignable clauses* for loops can tell what might be modified

```c
@ assignable i, a[*];
```
Want to keep part of the context that is *unmodified* by loop

**assignable** clauses for loops can tell what might be modified

```c
@ assignable i, a[*];
```
**Example with Improved Invariant Rule**

```java
int i = 0;
while (i < a.length) {
    a[i] = 1;
    i++;
}
```
Example with Improved Invariant Rule

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

```java
int i = 0;
while (i < a.length) {
    a[i] = 1;
    i++;
}
```
Example with Improved Invariant Rule

Precondition: \( a \neq \text{null} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a\text{.length }\rightarrow a[x] = 1) \)
Example with Improved Invariant Rule

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

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

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

**Loop invariant:** $0 \leq i \land i \leq a.length$
Example with Improved Invariant Rule

Precondition: \( a \neq \text{null} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a.length \rightarrow a[x] = 1) \)

Loop invariant: \( 0 \leq i \land i \leq a.length \land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
Example with Improved Invariant Rule

Precondition: \( a \neq \text{null} \)

\begin{verbatim}
int i = 0;
while (i < a.length) {
    a[i] = 1;
    i++;
}
\end{verbatim}

Postcondition: \( \forall \text{int } x; (0 \leq x < a.length \rightarrow a[x] = 1) \)

Loop invariant: \( 0 \leq i \land i \leq a.length \land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
Example with Improved Invariant Rule

Precondition: \( a \neq \text{null} \) & \( \text{ClassInv} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a.\text{length} \rightarrow a[x] = 1) \)

Loop invariant: \( 0 \leq i \land i \leq a.\text{length} \)
\( \land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
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

∀ int x;
    (n \equiv x \land x \geq 0 \rightarrow
        [i = 0; r = 0;
            while (i<n) { i = i + 1; r = r + i;}
            r=r+r-n;
        ]r \equiv?)

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

\[ \forall \text{int } x; \]
\[ (n \div x \land x \geq 0 \rightarrow \]
\[ [i = 0; r = 0; \]
\[ \text{while } (i < n) \{ i = i + 1; r = r + i; \} \]
\[ r = r + r - n; \]
\[ ]r \div x \ast x) \]

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

Solution:

@ loop_invariant
@ \quad i \geq 0 \&\& \; 2 \ast r == i \ast (i + 1) \&\& \; i \leq n;
@ assignable \; i, \; r;

File: Loop2.java
Example

∀ int x;
  \((n \div x \land x \geq 0 \to \]
  \[i = 0; r = 0;
  \textbf{while} (i<n) \{ i = i + 1; r = r + i;\}
  r=r+r-n;
\]
  \[r \div x \times x)\]

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

Solution:

@ \texttt{loop_invariant}
@ \texttt{i>0 \&\& 2*r == i*(i + 1) \&\& i <= n;}
@ \texttt{assignable i, r;}

File: 

File: Loop2.java
Example

∀ int x;
    (n ≥ x ∧ x ≥ 0 → 
    [ i = 0; r = 0;
        while (i<n) { i = i + 1; r = r + i;}
        r=r+r-n;
    ] r ≥ 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 on the slides assumes that assignable is correct. With assignable \nothing; e.g., one can prove nonsense.
- The invariant rule in 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 on the slides *assumes* that assignable is correct. With `assignable \nothing;` e.g., one can prove nonsense.
- The invariant rule in 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

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
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
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
```java
@ decreasing
```
Find a decreasing integer term \( v \) (called \textit{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 \texttt{diverges true;}
- Add directive \texttt{decreasing } \( v \); to loop invariant
- KeY creates suitable invariant rule and PO (with \( \langle \ldots \rangle \phi \))

Example: The \texttt{array} loop

\@ \texttt{decreasing a.length - i;
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
Side effects in loop guards

Find a postcondition:

```c
int x, y;
// ...
while( x-- != ++y );
```

Note: Loop guards may have side effects. Hence: Evaluate them in a modality.

Invariant rule with side effects

\[
\text{sideEffectLI} \quad \begin{align*}
\Gamma &\Rightarrow \mathcal{U} \text{Inv}, \Delta \\
\text{Inv}, b &\equiv \text{TRUE} < 4 > [x=b;]x \equiv \text{TRUE} \Rightarrow [p]\text{Inv} \\
\text{Inv}, b &\equiv \text{FALSE} \Rightarrow [\pi \omega]\phi \\
\Gamma &\Rightarrow \mathcal{U}[\pi \text{while}(b) p \omega]\phi, \Delta
\end{align*}
\]
Side effects in loop guards

Find a postcondition:

```c
int x, y;
// ...
while( x-- ! = ++y );
```

**Note:** Loop guards may have side effects.

**Hence:** Evaluate them in a modality.

Invariant rule with side effects

\[
\text{sideEffectLI} \quad \Gamma \Rightarrow U \text{Inv}, \Delta \\
\text{Inv}, b \models \text{TRUE} < 4 > [x=b]; x \models \text{TRUE} \Rightarrow [p] \text{Inv} \\
\text{Inv}, b \models \text{FALSE} \Rightarrow [\pi \omega] \phi \\
\Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta
\]
Side effects in loop guards

Find a postcondition:

```c
int x, y;
// ...
while ( x-- != ++y );
```

**Note:** Loop guards may have side effects.
**Hence:** Evaluate them in a modality.

Invariant rule with side effects

\[
\Gamma \Rightarrow U \text{Inv}, \Delta \\
\begin{align*}
\text{Inv}, \ b &\doteq \text{TRUE} \ < 4 > \ [x=b;]x &\doteq \text{TRUE} \Rightarrow [p]\text{Inv} \\
\text{Inv}, \ b &\doteq \text{FALSE} \Rightarrow [\pi \omega]\phi \\
\text{sideEffectLI} &
\end{align*}
\Rightarrow U[\pi \text{while}(b) \ p \omega]\phi, \Delta
\]

(initially valid) (preserved) (use case)
Side effects in loop guards

Find a postcondition:

```c
int x, y;
// ...
while( x-- != ++y );
```

**Note:** Loop guards may have side effects.

**Hence:** Evaluate them in a modality.

**Invariant rule with side effects**

\[
\begin{align*}
\text{sideEffectLI} & \quad \Gamma \Rightarrow U \text{Inv}, \Delta \\
\text{Inv}, [x=b;]x \equiv \text{TRUE} & \Rightarrow [\rho] \text{Inv} \\
\text{Inv}, [x=b;]x \equiv \text{FALSE} & \Rightarrow [\pi \omega] \phi \\
\Gamma & \Rightarrow U[\pi \text{while}(b) \rho \omega] \phi, \Delta
\end{align*}
\]

(initially valid)
(preserved)
(use case)
Side effects in loop guards

Find a postcondition:

```c
int x, y;
// ...
while( x-- != ++y );
```

**Note:** Loop guards may have side effects.
**Hence:** Evaluate them in a modality.

**Invariant rule with side effects**

\[
\begin{align*}
\Gamma & \Rightarrow \mathcal{U} \text{Inv}, \Delta \\
\text{Inv}, \ [x=b \ ; \ ]x & \vdash \text{TRUE} \Rightarrow \ [x=b \ ; \ p] \text{Inv} \\
\text{Inv}, \ [x=b \ ; \ ]x & \vdash \text{FALSE} \Rightarrow \ [\pi \ \omega] \phi \\
\text{sideEffectLI} & \Rightarrow \Gamma \Rightarrow \mathcal{U}[\pi \ \text{while}(b) \ p \ \omega] \phi, \Delta
\end{align*}
\]

(Initially valid)
(Preserved)
(Use case)
Loops and Abrupt Completion

Rule `loopInvariant` requires normal, structural control flow (loop body always fully executed; run continues after loop)

Non-structural control flow in Java

- `return`
- `break`
- `continue`
- `throw`

make loop body terminate abruptly.

Solution

Transform non-standard control flow into standard control-flow and with marker variables.
Loops and Abrupt Completion

Original loop body $p$

```java
if (x == 0) break;
if (x == 1) return 42;
if (x == 2) continue;
if (x == 3) throw e;
if (x == 4) x = -1;
```

Encoded loop body $\hat{p}$

```java
loopBody: { try {
    BREAK = RETURN = false;
    EXCEPTION = null;
    if (x == 0) { BREAK = true;
                   break loopBody; }
    if (x == 1) { res = 42;
                  RETURN = true;
                  break loopBody; }
    if (x == 2) break loopBody;
    if (x == 3) throw e;
    if (x == 4) x = -1;
} catch (Throwable e) { EXC = e; }}
```
Loops and Abrupt Completion

Original loop body $p$

```plaintext
if(x == 0) break;
if(x == 1) return 42;
if(x == 2) continue;
if(x == 3) throw e;
if(x == 4) x = -1;
```

Encoded loop body $\hat{p}$

```plaintext
loopBody: { try {
  \text{BREAK} = \text{RETURN} = \text{false};
  \text{EXCEPTION} = \text{null};
  \text{if}(x == 0) { \text{BREAK}=\text{true};
    break loopBody; }
  \text{if}(x == 1) { \text{RES}=42;
    \text{RETURN}=\text{true};
    break loopBody; }
  \text{if}(x == 2) break loopBody;
  \text{if}(x == 3) throw e;
  \text{if}(x == 4) x = -1;
} catch(Throwable e) { \text{EXC} = e; }
```
Loops and Abrupt Termination

Invariant rule with abrupt termination (using translation \(\hat{\cdot}\))

\[
\begin{align*}
\text{loopInvariant} & : \quad \Gamma \implies \mathcal{U} \text{Inv}, \Delta \\
\quad \text{Inv, } b \doteq \text{TRUE} & \implies [p] \text{Inv} \\
\quad \text{Inv, } b \doteq \text{FALSE} & \implies [\pi \omega] \phi \\
\quad \Gamma & \implies \mathcal{U}[\pi \text{while} (b) p \omega] \phi, \Delta
\end{align*}
\]

(Initially valid)

(Preserved)

(Use case)

where \(\psi\) is the formula:

\[
(\text{Exc} \neq \text{null} \rightarrow [\pi \text{throw EXCEPTION}; \omega] \phi)
\]

\[
\land (\text{BREAK} \doteq \text{TRUE} \rightarrow [\pi \omega] \phi)
\]

\[
\land (\text{RETURN} = \text{TRUE} \rightarrow [\pi \text{return res}; \omega] \phi)
\]

\[
\land (\text{NORMAL} \rightarrow \text{Inv})
\]

with \(\text{NORMAL} \equiv \text{BREAK} \doteq \text{FALSE} \land \text{RETURN} \doteq \text{FALSE} \land \text{Exc} \doteq \text{null}\)
Loops and Abrupt Termination

Invariant rule with abrupt termination (using translation $\hat{\cdot}$)

\[
\begin{align*}
\Gamma & \implies U \text{Inv}, \Delta & \text{(initially valid)} \\
\text{Inv}, \ b \doteq \text{TRUE} & \implies [\hat{p}]\psi & \text{(preserved)} \\
\text{Inv}, \ b \doteq \text{FALSE} & \implies [\pi \omega]\phi & \text{(use case)} \\
\end{align*}
\]

where $\psi$ is the formula

\[
\begin{align*}
& (\text{Exc} \neq \text{null} \rightarrow [\pi \text{throw EXCEPTION}; \omega]\phi) \\
& \land (\text{BREAK} \doteq \text{TRUE} \rightarrow [\pi \omega]\phi) \\
& \land (\text{RETURN} = \text{TRUE} \rightarrow [\pi \text{return res}; \omega]\phi) \\
& \land (\text{NORMAL} \rightarrow \text{Inv})
\end{align*}
\]

with $\text{NORMAL} \equiv \text{BREAK} \doteq \text{FALSE} \land \text{RETURN} \doteq \text{FALSE} \land \text{Exc} \doteq \text{null}$
Loop Invariant – Conclusion

Is a difficult subject.
shows that real prog language is a challenge
Many technical non-trivial tricks.
A rule that puts together

1. considering assignable clauses
2. side effects in loop guards
3. abrupt termination

is in chapter 3.
Further reading: KeY book Ch. 15 ??
New developments: Loop scope rule, Loop contracts