Formal Systems II: Applications

Functional Verification of Java Programs: Java Dynamic Logic

Bernhard Beckert · Mattias Ulbrich | SS 2017
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 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 \( \psi \rightarrow [\alpha]\phi \)
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

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

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

- Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x \equiv \text{val}) \leftrightarrow (\langle q \rangle x \equiv \text{val}))\]

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

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

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

- Program formulas can appear nested

\[\forall \text{int } \text{val}; ((\langle p \rangle x \doteq \text{val} \rangle \leftrightarrow (\langle q \rangle x \doteq \text{val}))\]

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

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

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

- 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

\[(balance \geq c \& amount > 0) \rightarrow \langle charge(amount); \rangle balance > c\]

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

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

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

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

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

\begin{verbatim}
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]))

\end{verbatim}
Variables

- 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 J A V A C A R D D L 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
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

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

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

Semantics

Same as the formula

\[ (\psi_1 \& \ldots \& \psi_m) \Rightarrow (\phi_1 \mid \ldots \mid \phi_n) \]
Sequents and their Semantics

Syntax

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

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

Semantics

Same as the formula

\[ (\psi_1 \& \cdots \& \psi_m) \implies (\phi_1 | \cdots | \phi_n) \]
Sequent Rules

General form

\[
\Gamma_1 \rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \rightarrow \Delta_r \\
\Gamma \rightarrow \Delta
\]

Premisses

Conclusion

\((r = 0 \text{ possible: closing rules})\)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

\[
\begin{array}{c}
\text{Premisses} \\
\Gamma_1 \implies \Delta_1 \quad \cdots \quad \Gamma_r \implies \Delta_r \\
\hline
\Gamma \implies \Delta \\
\end{array}
\]

\( (r = 0 \text{ possible: closing rules}) \)

Soundness
If all premisses are valid, then the conclusion is valid

Use in practice
Goal is matched to conclusion
Sequent Rules

General form

Premisses

\[ \Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \]

Conclusion

\[ \Gamma \Rightarrow \Delta \]

\((r = 0 \text{ possible: closing rules})\)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

\[
\begin{array}{c}
\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r}{\Gamma \Rightarrow \Delta}
\end{array}
\]

(rule_name)

(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

\[
\text{not_left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, ! A \Rightarrow \Delta}
\]

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

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

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

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

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

not_left

\[
\Gamma \Rightarrow A, \Delta \\
\Rightarrow \Gamma, ! A \Rightarrow \Delta
\]

imp_left

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

close_goal

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

close_by_true

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

all_left

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

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

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

- **imp_left**
  \[
  \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
  \]

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

- **not_left**
  \[
  \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, ! A \Rightarrow \Delta}
  \]

- **imp_left**
  \[
  \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
  \]

- **close_goal**
  \[
  \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow \Delta}
  \]

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

- **not_left**
  \[ \Gamma \Rightarrow A, \Delta \]
  \[ \Gamma, \forall A \Rightarrow \Delta \]

- **imp_left**
  \[ \Gamma \Rightarrow A, \Delta \]
  \[ \Gamma, B \Rightarrow \Delta \]
  \[ \Gamma, A \rightarrow B \Rightarrow \Delta \]

- **close_goal**
  \[ \Gamma, A \Rightarrow A, \Delta \]

- **close_by_true**
  \[ \Gamma \Rightarrow true, \Delta \]

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

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
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

```java
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

```java
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

\[
\begin{align*}
1: \{ & \text{try}\{ \pi \} \text{ i=0; } j=0; \} \text{ finally}\{ k=0; \} \}
\end{align*}
\]

- Passive prefix: \( \pi \)
- Active statement: \( i=0; \)
- Rest: \( \omega \)

-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

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

- 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

\[
\Gamma, B = true \implies \langle p \ \omega \rangle \phi, \Delta \\
\Gamma, B = false \implies \langle q \ \omega \rangle \phi, \Delta
\]

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

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

\[
\begin{align*}
\Gamma, B = \text{true} & \Rightarrow \langle p \omega \rangle \phi, \Delta \\
\Gamma, B = \text{false} & \Rightarrow \langle q \omega \rangle \phi, \Delta \\
\Gamma & \Rightarrow \langle \text{if } (B) \{ p \} \text{ else } \{ q \} \omega \rangle \phi, \Delta
\end{align*}
\]

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

\[
\begin{align*}
\Gamma & \Rightarrow \langle v=y; y=y+1; x=v; \omega \rangle \phi, \Delta \\
\Gamma & \Rightarrow \langle x=y++; \omega \rangle \phi, \Delta
\end{align*}
\]

Simple assignment rule

\[
\begin{align*}
\Gamma & \Rightarrow \{ \text{loc := val} \} \langle \omega \rangle \phi, \Delta \\
\Gamma & \Rightarrow \langle \text{loc=val; } \omega \rangle \phi, \Delta
\end{align*}
\]
Rules for Symbolic Program Execution

If-then-else rule

\[
\Gamma, B = true \Rightarrow \langle p \omega \rangle \phi, \Delta \quad \Gamma, B = false \Rightarrow \langle q \omega \rangle \phi, \Delta
\]

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

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

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

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

Simple assignment rule

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

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

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

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

Updates
syntactic elements in the logic – (explicit substitutions)

Elementary Updates

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

where
- \( loc \) is a program variable
- \( val \) is an expression type-compatible with \( loc \)

Parallel Updates

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

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

Beckert, Ulbrich – Formal Systems II: Applications  SS 2017  20/39
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)

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

\[ ... \]

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

\[ ... \]

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

\[ ... \]

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

\[ ... \]

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

\[ ... \]

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

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

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

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

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

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

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

\[ \vdots \]

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

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

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

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

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

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

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

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

\[
x < y \quad \Rightarrow \quad x < y \\
\vdots \\
x < y \quad \Rightarrow \quad \{x:=y \parallel y:=x\} y < x \\
\vdots \\
x < y \quad \Rightarrow \quad \{t:=x \parallel x:=y \parallel y:=x\} y < x \\
\vdots \\
x < y \quad \Rightarrow \quad \{t:=x \parallel x:=y\} \{y:=t\} y < x \\
\vdots \\
x < y \quad \Rightarrow \quad \{t:=x\} \{x:=y\} \langle y=t; \rangle y < x \\
\vdots \\
x < y \quad \Rightarrow \quad \{t:=x\} \langle x=y; \ y=t; \rangle y < x \\
\vdots \\
\Rightarrow \quad x < y \rightarrow \langle \text{int} \ t=x; \ x=y; \ y=t; \rangle y < x
\]
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

$\forall a, i, v. \ select(store(a, i, v), i) = v$

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

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 $\textit{Array}(\mathbb{I}, \mathbb{V})$

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

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

**Axioms**

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

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

**Intuition**

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

An abstract datatype

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 \( \mathcal{D}(\mathbb{I}) \rightarrow \mathcal{D}(\mathbb{V}) \)

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

Photo by “null0” (www.flickr.com/photos/null0/272015955)
Program State Representation

Local program variables
Modeled as non-rigid constants

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

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)
extc 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: $\mathbb{I} = \text{Object} \times \text{Field}, \forall = \text{Any}$

- $\text{heap}$: $\text{Heap}$ (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: $\mathbb{I} = \text{Object} \times \text{Field}$, $\mathbb{V} = \text{Any}$

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

### 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
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

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

unwindLoop

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

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

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

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

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

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

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

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 $1 \times$
- 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)
Symbolic execution of loops: unwind

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

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

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

**unwindLoop**

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

---

**How to handle a loop with...**

- 0 iterations? Unwind $1 \times$
- 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

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

How to handle a loop with…

- 0 iterations? Unwind 1 ×
- 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 \textit{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

\[
\text{loopInvariant} \quad \frac{\Gamma \Rightarrow U Inv, \Delta}{\Gamma \Rightarrow U[\pi \text{while} (b) \ p \omega] \phi, \Delta} \quad \text{(initially valid)} \\
\quad \frac{Inv, \ b \neq \text{TRUE} \Rightarrow [p] Inv}{(preserved)} \\
\quad \frac{Inv, \ b \neq \text{FALSE} \Rightarrow [\pi \omega] \phi}{(use \ case)}
\]
Loop Invariants Cont’d

Idea behind loop invariants

- A formula \( \textit{Inv} \) whose validity is \textit{preserved} by loop guard and body
- \textit{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 \textit{afterwards}
- Encode the desired \textit{postcondition} after loop into \( \textit{Inv} \)

Basic Invariant Rule

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

Idea behind loop invariants

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

Basic Invariant Rule

$\Gamma \Rightarrow \Upsilon \mathit{Inv}, \Delta$ (initially valid)

$\mathit{inv}, b \Downarrow \mathit{TRUE} \Rightarrow [p] \mathit{Inv}$ (preserved)

$\mathit{inv}, b \Downarrow \mathit{FALSE} \Rightarrow [\pi \omega] \phi$ (use case)

$\Gamma \Rightarrow \Upsilon [\pi \mathit{while}(b) \ p \ \omega] \phi, \Delta$
Loop Invariants Cont’d

Idea behind loop invariants

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

Basic Invariant Rule

\[
\Gamma \Rightarrow U \begin{array}{c} \text{Inv}, \Delta \\ \text{Inv}, \; b \doteq \text{TRUE} \Rightarrow [p] \text{Inv} \\ \text{Inv}, \; b \doteq \text{FALSE} \Rightarrow [\pi \omega] \phi \end{array} \ \\
\Gamma \Rightarrow U[\pi \text{while}(b) \; p \; \omega] \phi, \Delta
\]
Loop Invariants Cont’d

Idea behind loop invariants

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

Basic Invariant Rule

\[
\begin{array}{c}
\Gamma \Rightarrow \mathcal{U} \mathbf{Inv}, \Delta & \text{(initially valid)} \\
Inv, \ b \vdash \text{TRUE} \Rightarrow [p]Inv & \text{(preserved)} \\
Inv, \ b \vdash \text{FALSE} \Rightarrow [\pi \omega]\phi & \text{(use case)} \\
\Gamma \Rightarrow \mathcal{U}[\pi \text{while} (b) \ p \omega]\phi, \Delta
\end{array}
\]
Loop Invariants Cont’d

Basic Invariant Rule: Problem

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

- 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 \text{Inv}
Basic Invariant Rule: Problem

\[ \Gamma \Rightarrow U \text{Inv}, \Delta \]  
(initially valid)

\[ \text{Inv}, b \models \text{TRUE} \Rightarrow [p]\text{Inv} \]  
(preserved)

\[ \text{Inv}, b \models \text{FALSE} \Rightarrow [\pi \omega]\phi \]  
(use case)

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

- 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

\[ \Gamma \Rightarrow U \text{Inv}, \Delta \]

\( \text{Inv, } b \models \text{TRUE} \Rightarrow [p] \text{Inv} \)  (initially valid)

\( \text{Inv, } b \models \text{FALSE} \Rightarrow [\pi \omega] \phi \)  (preserved)

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

- Context \( \Gamma, \Delta, U \) must be omitted in 2nd and 3rd premise
- \textbf{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

\[
\begin{align*}
\text{loopInvariant} & \quad \Gamma \Rightarrow U\ Inv, \Delta \\
& \quad \text{Initially valid} \\
\text{Inv, } b \equiv \text{TRUE} \Rightarrow [p] Inv \\
& \quad \text{Preserved} \\
\text{Inv, } b \equiv \text{FALSE} \Rightarrow [\pi \omega] \phi \\
& \quad \text{Use case} \\
\Gamma \Rightarrow U[\pi \text{while}(b) \ p \omega] \phi, \Delta
\end{align*}
\]

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

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

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

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

**Loop invariant:** \( 0 \leq i \& i \leq a\text{.length} \)
Example

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

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

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

Loop invariant: \( 0 \leq i \) & \( i \leq a\text{.length} \)

& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \doteq 1) \)
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: \( 0 \leq i \& i \leq a\text{.length} \)
\& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
\& \( a \neq \text{null} \)
Example

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

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

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

Loop invariant: \( 0 \leq i \& i \leq a.\text{length} \)

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

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

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

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

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

$\& \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \doteq 1)$
Example in JML/Java – Loop.java

```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

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

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

Solution:

@ \textbf{loop_invariant}
@ i \geq 0 \land 2 \ast r = i \ast (i + 1) \land i \leq n;
@ \textbf{assignable} \ i, \ r;

File: Loop2.java
Example

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

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

Solution:

@ \textbf{loop-invariant}
@ \textit{i} \geq 0 \&\& 2 \times r \equiv i \times (i + 1) \&\& i \leq n;
@ \textbf{assignable} i, r;

File: Loop2.java
Example

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

@ loop_invariant
@ \ i \geq 0 \land 2 \times r = i \times (i + 1) \land i \leq n;
@ assignable \ i, \ r;

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 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 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 \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 array loop

\[ \@ \text{decreasing} \]
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 \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 array loop

\( @ \texttt{decreasing} \)
Total Correctness

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

Add the following premisses to the invariant rule:
- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- $v$ is strictly decreased by the loop body

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

Example: The array loop
- `@ decreasing a.length - i;`
Total Correctness

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

Add the following premisses to the invariant rule:
- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- $v$ is strictly decreased by the loop body

Proving termination in JML/Java

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

Example: The \texttt{array} loop

\@ \texttt{decreasing} a.length - i;

Files:
- LoopT.java
- Loop2T.java