Formal Methods for Java

Lecture 13: Dynamic Logic

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

June 14, 2017

The KeY-Project

- Theorem Prover
- Developed at University of Karlsruhe
- `http://www.key-project.org/`.

- Theory specialized for Java(Card).
- Can generate proof-obligations from JML specification.
- Underlying theory: Sequent Calculus + Dynamic Logic

Dynamic Logic

Dynamic logic extends predicate logic by

- $[\alpha]\phi$
- $\langle\alpha\rangle\phi$

where α is a program and ϕ a sub-formula.

The meaning is as follows:

- $[\alpha]\phi$: after all terminating runs of program α formula ϕ holds.
- $\langle\alpha\rangle\phi$: after some terminating run of program α formula ϕ holds.

Comparison with Hoare Logic

The sequent $\phi \Longrightarrow [\alpha]\psi$ corresponds to partial correctness of the Hoare formula:

$$\{\phi\}\alpha\{\psi\}$$

If α is deterministic, $\phi \Longrightarrow \langle\alpha\rangle\psi$ corresponds to total correctness.

Examples

- $[\{\}]\phi \equiv \phi$
- $\langle\{\}\rangle\phi \equiv \phi$
- $[\textbf{while}(\textbf{true})\{\}]\phi \equiv \textbf{true}$
- $\langle\textbf{while}(\textbf{true})\{\}\rangle\phi \equiv \textbf{false}$
- $[x = x + 1;]x \geq 4 \equiv x + 1 \geq 4$
- $[x = t;]\phi \equiv \phi[t/x]$
- $[\alpha_1\alpha_2]\phi \equiv [\alpha_1][\alpha_2]\phi$

How can we use equivalences in Sequent Calculus?

Add the rule $\dfrac{\Gamma[\psi/\phi] \Longrightarrow \Delta[\psi/\phi]}{\Gamma \Longrightarrow \Delta}$, where $\phi \equiv \psi$.

This is similar to applyEq.

Dynamic Logic is Modal Logic

- $\langle\alpha\rangle\phi \equiv \neg[\alpha]\neg\phi$
- $[\alpha]\phi \equiv \neg\langle\alpha\rangle\neg\phi$

Furthermore:

- if ϕ is a tautology, so is $[\alpha]\phi$
- $[\alpha](\phi \to \psi) \to ([\alpha]\phi \to [\alpha]\psi)$

Remark: For deterministic programs also the reverse holds

$$([\alpha]\phi \to [\alpha]\psi) \to [\alpha](\phi \to \psi)$$

Termination and Deterministic Programs

How can we express that program α must terminate?

$$\langle\alpha\rangle\textbf{true}$$

This can be used to relate $[\alpha]$ and $\langle\alpha\rangle$:

$$\langle\alpha\rangle\phi \equiv [\alpha]\phi \wedge \langle\alpha\rangle\textbf{true}$$

Updates in KeY

The formula $\langle \texttt{i} = t; \alpha \rangle \phi$ is rewritten to

$$\{\texttt{i} := t\}\langle \alpha \rangle \phi$$

Formula $\{\texttt{i} := t\}\phi$ is true, iff
ϕ holds in a state, where the program variable $\texttt{i}$ has the value denoted by the term t.
Here:

- $\texttt{i}$ is a program variable (non-rigid function).
- t is a term (may contain logical variables).
- ϕ a formula

Simplifying Updates

If ϕ contains no modalities, then $\{x := t\}\phi$ is the substitution $\phi[t/x]$ (every occurence of x is changed to t).

A double update $\{x_1 := t_1, x_2 := t_2\}\{x_1 := t'_1, x_3 := t'_3\}\phi$ is automatically rewritten to

$$\{x_1 := t'_1[t_1/x_1, t_2/x_2], x_2 := t_2, x_3 := t'_3[t_1/x_1, t_2/x_2]\}\phi$$

Example: $\langle\{\mathtt{i} = \mathtt{j}; \mathtt{j} = \mathtt{i} + 1\}\rangle \mathtt{i} = \mathtt{j}$

$$
\begin{aligned}
&\langle\{\mathtt{i} = \mathtt{j}; \mathtt{j} = \mathtt{i} + 1\}\rangle \mathtt{i} = \mathtt{j} \\
\equiv&\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i+1}\}\mathtt{i} = \mathtt{j} \\
\equiv&\{\mathtt{i} := \mathtt{j}, \mathtt{j} := \mathtt{j} + 1\}\mathtt{i} = \mathtt{j} \\
\equiv&\mathtt{j} = \mathtt{j} + 1 \\
\equiv&\textbf{false}
\end{aligned}
$$

or alternatively

$$
\begin{aligned}
&\langle\{\mathtt{i} = \mathtt{j}; \mathtt{j} = \mathtt{i} + 1\}\rangle \mathtt{i} = \mathtt{j} \\
\equiv&\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i+1}\}\mathtt{i} = \mathtt{j} \\
\equiv&\{\mathtt{i} := \mathtt{j}\}\mathtt{i} = \mathtt{i} + 1 \\
\equiv&\mathtt{j} = \mathtt{j} + 1 \\
\equiv&\textbf{false}
\end{aligned}
$$

Rules for Java Dynamic Logic

- $\langle\{\texttt{i} = \texttt{j}; \ldots\}\rangle\phi$ is rewritten to:
  $\{\texttt{i} := \texttt{j}\}\langle\{\ldots\}\rangle\phi$.
- $\langle\{\texttt{i} = \texttt{j} + \texttt{k}; \ldots\}\rangle\phi$ is rewritten to:
  $\{\texttt{i} := \texttt{j} + \texttt{k}\}\langle\{\ldots\}\rangle\phi$.
- $\langle\{\texttt{i} = \texttt{j} + +; \ldots\}\rangle\phi$ is rewritten to:
  $\langle\{\textbf{int}\ \texttt{j\_0}; \texttt{j\_0} = \texttt{j}; \texttt{j} = \texttt{j} + 1; \texttt{i} = \texttt{j\_0}; \ldots\}\rangle\phi$.
- $\langle\{\textbf{int}\ \texttt{k}; \ldots\}\rangle\phi$ is rewritten to:
  $\langle\{\ldots\}\rangle\phi$ and $\texttt{k}$ is added as new program variable.

Proving Programs with Loops

Given a simple loop:

$$\langle\{\textbf{while}(\mathtt{n} > 0)\,\mathtt{n}\texttt{--};\}\rangle \mathtt{n} = 0$$

How can we prove that the loop terminates for all $\mathtt{n} \geq 0$ and that $\mathtt{n} = 0$ holds in the final state?

Method (1): Induction

To prove a property $\phi(x)$ for all $x \geq 0$ we can use induction:

- Show $\phi(0)$.
- Show $\phi(x) \implies \phi(x+1)$ for all $x \geq 0$.

This proves that $\forall x \, (x \geq 0 \rightarrow \phi(x))$ holds.

The rule int_induction

The KeY-System has the rule int_induction

$$\frac{\Gamma \Longrightarrow \Delta, \phi(0) \quad \Gamma \Longrightarrow \Delta, \forall X(X \geq 0 \land \phi(X) \to \phi(X+1)) \\ \Gamma, \forall X(X \geq 0 \to \phi(X)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

The three goals are:

- Base Case: $\Longrightarrow \phi(0)$
- Step Case: $\Longrightarrow \forall X(X \geq 0 \land \phi(X) \to \phi(X+1))$
- Use Case: $\forall X(X \geq 0 \to \phi(X)) \Longrightarrow$

Method(2): Loop Invariants with Variants

Induction proofs are very difficult to perform for a loop

$$\langle\{\textbf{while}(\mathit{COND})\,\mathit{BODY};\ldots\}\rangle\phi$$

The KeY-system supports special rules for while loops using invariants and variants.

The rule while_invariant_with_variant_dec

The rule while_invariant_with_variant_dec takes an invariant *inv*, a modifies set $\{m_1, \ldots, m_k\}$ and a variant v. The following cases must be proven.

- Initially Valid: $\Longrightarrow inv \land v \geq 0$
- Body Preserves Invariant:

$$\Longrightarrow \{m_1 := x_1 \| \ldots \| m_k := x_k\}(inv \land [\{b = COND;\}]b = \mathbf{true} \\ \rightarrow \langle BODY \rangle inv$$

- Use Case:

$$\Longrightarrow \{m_1 := x_1 \| \ldots \| m_k := x_k\}(inv \land [\{b = COND;\}]b = \mathbf{false} \\ \rightarrow \langle \ldots \rangle \phi$$

- Termination:

$$\Longrightarrow \{m_1 := x_1 \| \ldots \| m_k := x_k\}(inv \land v \geq 0 \land [\{b = COND;\}]b = \mathbf{true} \\ \rightarrow \{old := v\}\langle BODY \rangle v \leq old \land v \geq 0$$

Rigid vs.Non-Rigid Functions vs. Variables

KeY distinguishes the following symbols:

- Rigid Functions: These are functions that do not depend on the current state of the program.
  - $+, -, *$: *integer* $\times$ *integers* $\rightarrow$ *integer* (mathematical operations)
  - $0, 1, \ldots$: *integer*, *TRUE*, *FALSE* : *boolean* (mathematical constants)
- Non-Rigid Functions: These are functions that depend on current state.
  - $\cdot[\cdot] : \top \times int \rightarrow \top$ (array access)
  - `.next` : $\top \rightarrow \top$ if `next` is a field of a class.
  - `i`, `j` : $\top$ if `i`, `j` are program variables.
- Variables: These are logical variables that can be quantified. Variables may not appear in programs.
  - x, y, z

Example

$$\forall x.\mathtt{i} = x \rightarrow \langle\{\mathit{while}(\mathtt{i} > 0)\{\mathtt{i} = \mathtt{i} - 1;\}\}\rangle\mathtt{i} = 0$$

- 0,1,$-$ are rigid functions.
- $>$ is a rigid relation.
- $\mathtt{i}$ is a non-rigid function.
- x is a logical variable.

Quantification over $\mathtt{i}$ is not allowed and x must not appear in a program.

Builtin Rigid Functions

- $+, -, *, /, \%$, *jdiv*, *jmod*: operations on *integer*.
- $\ldots, -1, 0, 1, \ldots$, *TRUE*, *FALSE*, *null*: constants.
- (A) for any type A: cast function.
- $A :: get$ gives the n-th object of type A.