Formal Methods for Java

Lecture 12: Dynamic Logic

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

December 7, 2011

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

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.
  - $+, -, * : \mathit{integer} \times \mathit{integers} \to \mathit{integer}$ (mathematical operations)
  - $0, 1, \ldots : \mathit{integer}$, $\mathit{TRUE}, \mathit{FALSE} : \mathit{boolean}$ (mathematical constants)
- Non-Rigid Functions: These are functions that depend on current state.
  - $\cdot[\cdot] : \top \times \mathit{int} \to \top$ (array access)
  - $.\mathtt{next} : \top \to \top$ if `next` is a field of a class.
  - $\mathtt{i}, \mathtt{j} : \top$ 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.
- `i` is a non-rigid function.
- x is a logical variable.

Quantification over `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.

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 `i` has the value denoted by the term t.
Here:

- `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 rewritten to $\phi[t/x]$.

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

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

or alternatively

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

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:

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

How can we prove that the loop terminates for all $\texttt{n} \geq 0$ and that $\texttt{n} = 0$ 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 $\phi(x) \Longrightarrow \phi(x+1)$ 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

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

The three goals are:

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

Method(2): Loop Invariants with Variants

Induction proofs are very difficult to perform for a loop

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

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:

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

- Use Case:

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

- Termination:

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

Case Study: Euklid's Algorithm

Java code to compute gcd of non-negative numbers:

```
public static int gcd(int a, int b) {
    while (a != 0 && b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return (a > b) ? a : b;
}
```

Lets prove it with KeY-System.

Specification

We first need a specification.

Definition (GCD)

Let a and b be natural numbers. A number d is the greatest common divisor (GCD) of a and b iff

1. $d|a$ and $d|b$
2. If $c|a$ and $c|b$, then $c|d$.

$d|a$ means d divides a.
$d|a :\Leftrightarrow \exists q. d * q = a$

JML Specification

The specifation can be converted to JML:

```
/*@
  @ requires a >= 0 && b >= 0;
  @ ensures \result >= 0;
  @ ensures (\exists int q; \result*q == a) &&
  @         (\exists int q; \result*q == b) &&
  @  (\forall int c;
  @       (\exists int q; c*q == a) && (\exists int q; c*q == b);
  @       (\exists int q; c*q == \result));
  @*/
public static int gcd(int a, int b)
```

So lets start proving . . .

Loop-Invariant

What is the loop invariant?

The algorithm changes *a* and *b*, but the gcd of *a* and *b* should stay the same.

In fact the set of common divisors of *a* and *b* never changes.
This suggests the following invariant:

$$\forall d.(d| \texttt{\textbackslash old}(a) \land d| \texttt{\textbackslash old}(b) \leftrightarrow d|a \land d|b)$$

In JML this can be specified as:

```
/*@ loop_invariant a >= 0 && b >= 0 &&
  @     (\forall int d; true;
  @     (\exists int q; \old(a) == q*d)
  @      && (\exists int q; \old(b) == q*d)
  @ <==>(\exists int q;  a == q*d) && (\exists int q; b == q*d)
  @     );
  @ assignable a, b;
  @ decreases a+b;
  @*/
```