### Automatic Validation of Transformation Rules for Java Verification against a Rewriting Semantics

**Ralf Sasse** 

University of Illinois at Urbana-Champaign

Joint Work with:

Wolfgang Ahrendt, Chalmers University of Technology, Göteborg

Andreas Roth, Universität Karlsruhe

## **KeY - Dynamic Logic**

KeY:

- Interactive Java source code prover.
- Based on Java Dynamic Logic.
- Dynamic Logic formula:

 $\langle \pi \rangle \phi$ 

Example:

 $\langle i=(j=i)*(i++); \rangle i \doteq j*j$ 

Sequent calculus: Dynamic Logic rules.

### **Example Rule**

Rule:

$$\frac{\Gamma \vdash \langle \text{typeof(e) } v_1 = e; \text{ typeof(n) } v_2 = n; \text{ } I = v_1 * v_2; \text{ } rs \rangle \phi, \Delta}{\Gamma \vdash \langle I = e * n; \text{ } rs \rangle \phi, \Delta}$$

Variables are schema variables.

Special case here: program transformation rule.

### **Rules Implemented as Taclets**

Rule:

$$\frac{\Gamma \vdash \langle \text{typeof(e) } v_1 = e; \text{ typeof(n) } v_2 = n; \text{ I} = v_1 * v_2; \text{ rs} \rangle \phi, \Delta}{\Gamma \vdash \langle \text{I} = e * n; \text{ rs} \rangle \phi, \Delta}$$

Implementing taclet:

find( $\langle I = e * n; rs \rangle$  b) replacewith( $\langle typeof(e) v_1 = e; typeof(n) v_2 = n; I = v_1 * v_2; rs \rangle$  b) varcond(new typeof(e) v<sub>1</sub>, typeof(n) v<sub>2</sub>)

# **Taclet Application**

Taclet:

 $\begin{aligned} & \text{find}(\langle I=e*n; rs \rangle \ b) \\ & \text{replacewith}(\langle \text{typeof(e)} \ v_1=e; \ \text{typeof(n)} \ v_2=n; \ I=v_1*v_2; rs \rangle \ b) \\ & \text{varcond}(\text{new typeof(e)} \ v_1, \text{typeof(n)} \ v_2) \end{aligned}$ 

Taclet applicable in this formula:

$$\langle i=(j=i)*(i++); \rangle i \doteq j*j$$

Result of that application:

(int eval1=(j=i); int eval2=i++; i=eval1\*eval2;)

$$\mathtt{i}\doteq\mathtt{j}*\mathtt{j}$$

#### Aim

Java Dynamic Logic Calculus: 480 rules.

Out of that: 210 program transformation rules.

Other formalisations of Java:
 e.g. Java semantics in rewriting logic.

Aim:

automated validation of transformation rules vs. rewriting logic Java semantics

# **Rewriting Logic**

- **Proof** Rewrite Theory:  $(\Sigma, E, \mathcal{R})$
- Logical and computational view.
- Rule:  $t \rightarrow t'$ ,
- Equation: t = t', (Church-Rosser and terminating).
- Rule application modulo equations.

# **Rewriting Logic -** $\mathcal{R}_{Java}$

- Rewriting Logic implementation: Maude.
- **Java semantics given in Maude, call it**  $\mathcal{R}_{Java}$ :
  - (executable) specification of Java,
  - interpreter for free.
- Used  $\mathcal{R}_{Java}$  version is a prototype.

### (Cross-)Validating transformation taclets

General form of a program transformation rule:

 $\frac{\Gamma \vdash \langle \Pi' \mathsf{ rs} \rangle \phi, \Delta}{\Gamma \vdash \langle \Pi \mathsf{ rs} \rangle \phi, \Delta}$ 

- Problem: II, II' schematic, cannot execute schematic code in  $\mathcal{R}_{Java}$ .
- Program transformation with  $\Pi$ ,  $\Pi'$  correct if for all instances  $\pi$ ,  $\pi'$  this holds:

$$<\pi,s> \stackrel{\mathcal{R}_{Java}}{\longrightarrow} s'$$

$$<\pi',s> \stackrel{\mathcal{R}_{Java}}{\longrightarrow} s''$$

### (Cross-)Validating transformation taclets

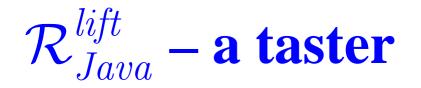
Program transformation with  $\Pi$ ,  $\Pi'$  correct if for all instances  $\pi$ ,  $\pi'$  this holds:

$$<\pi, s > \stackrel{\mathcal{R}_{Java}}{\longrightarrow} s'$$
 $<\pi', s > \stackrel{\mathcal{R}_{Java}}{\longrightarrow} s''$ 
 $s' == s''$ 

Idea: Lift semantics to allow execution of schematic code.

$$<\Pi,s> \stackrel{\mathcal{R}_{Java}^{lift}}{\longrightarrow} s'$$

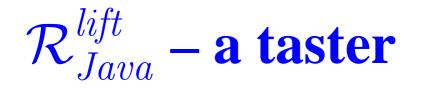
$$<\Pi', s > \stackrel{\mathcal{R}^{lift}_{Java}}{\longrightarrow} s''$$



Basic problem: evaluating schematic expressions.

Java expressions:

- depend on state
- may have side-effects



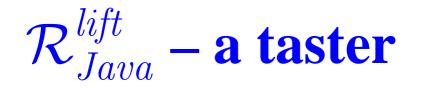
Basic problem: evaluating schematic expressions.

Java expressions:

- depend on state
- may have side-effects

schematic Java expressions:

- depend on symbolic state
- have unknown side-effects



Basic problem: evaluating schematic expressions.

Java expressions:

- depend on state
- may have side-effects

schematic Java expressions:

- depend on symbolic state
   modeled by 'snapshots'
- have unknown side-effects
  - modeled by 'extended conditional values' in memory

## **Snapshots**

- $\mathcal{R}_{Java}^{lift}$  uses symbolic configuration, consisting of: memory, environment, continuation, ...
- **e.g. symbolic memory:** [L1,V1] [L2,V2] rm
- Snapshots save relevant parts of configuration: memory, environment and current object.

### **Extended Conditional Values**

Treating schematic expressions e with unknown side effects and result:

- Side effect in general: change n locations  $L_1, \ldots, L_n$  to values  $V_1, \ldots, V_n$ .
- e executed in configuration s: location list L1(e, s) changed to value list V1(e, s).
- Execute e in configuration s: for all [L,V] in memory afterwards at location L the extended conditional value: L in Ll(e,s) ?? Vl(e,s) :: V
- Introduced into memory by operator walking through it. Finally sticks at symbolic memory rest rm.

### **Configuration Generation**

Still: types for schema variables too general for  $\mathcal{R}_{Java}^{lift}$ .

- e.g.: type *lefthandside* could be either of:
  - Iocal variable: add [x, L] to local environment,
  - static variable:
     add [x, L] to static environment,
  - attribute of the current object:
     add [x,L] to the current object's environment.
- In each case: add [L, V] to generic memory.
- Check all possible combinations, can be over 100 cases per taclet.
- Automated generation of start configurations.

#### **Results**

- Lifted semantics for concrete Java:  $\mathcal{R}_{Java}^{lift}$ , can now handle schematic Java!
- Program Transformation Taclets:
  - Automatically validated 56 of 210 transformation taclets,
  - could validate more, if original  $\mathcal{R}_{Java}$  was complete
- Daily automated use of this method validates transformation taclets in KeY every night. Run takes about 3 minutes.
- Found 3 unsounds taclets.

### Example

Actual KeY program transformation rule:

$$\frac{\Gamma \vdash \langle \mathsf{x} = \mathsf{y} ; \mathsf{y} = \mathsf{y} + \mathsf{l}; \mathsf{rs} \rangle \phi, \Delta}{\Gamma \vdash \langle \mathsf{x} = \mathsf{y} + \mathsf{i}; \mathsf{rs} \rangle \phi, \Delta}$$

### **Same Instantiations for Different SVs**

Program transformation (wrong)

$$x=y++i$$
  $\longrightarrow$   $x=yi$   $y=y+1i$ 

Instantiate x and y with a: a=a++; keeps a (according to Java Lang. Spec.) a=a; a=a+1; changes a (obviously).

Corrected program transformation:

$$x=y++i$$
  $\longrightarrow$   $v=yi$   $y=y+1i$   $x=vi$ 

#### **Future Work**

- $\mathcal{R}_{Java}$  with more features: check more different transformation taclets.
  - Specifically exception-handling!
- Extend scope and handle more than pure transformation taclets: e.g. branching rules.