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:

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

- Sequent calculus: Dynamic Logic rules.

Example Rule

Rule:

$$\frac{\Gamma \ \vdash \ \langle \mathsf{typeof(e)}\ \mathsf{v}_1 = \mathsf{e};\ \mathsf{typeof(n)}\ \mathsf{v}_2 = \mathsf{n};\ \mathsf{l} = \mathsf{v}_1 * \mathsf{v}_2;\ \mathsf{rs} \rangle\ \phi, \Delta}{\Gamma \ \vdash \ \langle \mathsf{l} = \mathsf{e} * \mathsf{n};\ \mathsf{rs} \rangle\ \phi, \Delta}$$

Variables are schema variables.

Special case here: *program transformation* rule.

Rules Implemented as Taclets

Rule:

$$\frac{\Gamma \ \vdash \ \langle \mathsf{typeof(e)}\ \mathsf{v}_1 = \mathsf{e};\ \mathsf{typeof(n)}\ \mathsf{v}_2 = \mathsf{n};\ \mathsf{l} = \mathsf{v}_1 * \mathsf{v}_2;\ \mathsf{rs}\rangle\ \phi, \Delta}{\Gamma \ \vdash \ \langle \mathsf{l} = \mathsf{e} * \mathsf{n};\ \mathsf{rs}\rangle\ \phi, \Delta}$$

Implementing taclet:

$\mathsf{find}\big(\langle \mathsf{l} = \mathsf{e} * \mathsf{n}; \mathsf{rs}\rangle\ \mathsf{b}\big)$

$\mathsf{replacewith}\big(\langle \mathsf{typeof(e)}\ \mathsf{v}_1 = \mathsf{e};\ \mathsf{typeof(n)}\ \mathsf{v}_2 = \mathsf{n};\ \mathsf{l} = \mathsf{v}_1 * \mathsf{v}_2; \mathsf{rs}\rangle\ \mathsf{b}\big)$

$\mathsf{varcond}\big(\mathsf{new}\ \mathsf{typeof(e)}\ \mathsf{v}_1, \mathsf{typeof(n)}\ \mathsf{v}_2\big)$

Taclet Application

Taclet:

$$\mathsf{find}\big(\langle \mathsf{l} = \mathsf{e} * \mathsf{n}; \mathsf{rs}\rangle\ \mathsf{b}\big)$$
$$\mathsf{replacewith}\big(\langle \mathsf{typeof(e)}\ \mathsf{v}_1 = \mathsf{e};\ \mathsf{typeof(n)}\ \mathsf{v}_2 = \mathsf{n};\ \mathsf{l} = \mathsf{v}_1 * \mathsf{v}_2; \mathsf{rs}\rangle\ \mathsf{b}\big)$$
$$\mathsf{varcond}\big(\mathsf{new\ typeof(e)}\ \mathsf{v}_1, \mathsf{typeof(n)}\ \mathsf{v}_2\big)$$

Taclet applicable in this formula:

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

Result of that application:

$$\langle \texttt{int eval1=(j=i); int eval2=i++; i=eval1*eval2;}\rangle$$
$$\texttt{i} \doteq \texttt{j} * \texttt{j}$$

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

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

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

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

$$s' == s''$$

(Cross-)Validating transformation taclets

Program transformation with Π, Π' correct if for all instances π, π' 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.

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

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

$$s' == s''$$

$\mathcal{R}^{lift}_{Java}$ – a taster

Basic problem: evaluating schematic expressions.

Java expressions:

- depend on state
- may have side-effects

$\mathcal{R}^{lift}_{Java}$ – a taster

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

$\mathcal{R}^{lift}_{Java}$ – a taster

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

- $\mathcal{R}^{lift}_{Java}$ 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 `Ll(`e, s`)` changed to value list `Vl(`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 $\mathcal{R}^{lift}_{Java}$.
- e.g.: type *lefthandside* could be either of:
  - local 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:
  $\mathcal{R}^{lift}_{Java}$, 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:

$$\frac{\Gamma \ \vdash \ \langle \mathsf{x = y}\ \texttt{;}\ \mathsf{y = y + }\texttt{1;}\ \mathsf{rs} \rangle\ \phi, \Delta}{\Gamma \ \vdash \ \langle \mathsf{x = y}\texttt{++;}\ \mathsf{rs} \rangle\ \phi, \Delta}$$

Same Instantiations for Different SVs

Program transformation (wrong)

$$\texttt{x=y++;} \leadsto \texttt{x=y; y=y+1;}$$

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:

$$\texttt{x=y++;} \leadsto \texttt{v=y; y=y+1; x=v;}$$

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.