# Introduction to JavaCard Dynamic Logic

Andreas Roth, Richard Bubel, Christian Engel

November 22, 2006

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination
  - **Exceptions** (`try` – `catch` – `finally`)

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination
  - **Exceptions** (`try` – `catch` – `finally`)
  - **Local jumps** `return`, `break`, `continue`

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination
  - **Exceptions** (`try` – `catch` – `finally`)
  - **Local jumps** `return`, `break`, `continue`
- **Aliasing**
  Different navigation expressions may be same object reference

$$I \models \texttt{o.age} \doteq 1 \;\texttt{->}\; \langle \texttt{u.age} = 2; \rangle \texttt{o.age} \doteq \texttt{u.age} \qquad \textbf{?}$$

  Depends on whether $I \models \texttt{o} \doteq \texttt{u}$

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination
  - **Exceptions** (`try` – `catch` – `finally`)
  - **Local jumps** `return`, `break`, `continue`
- **Aliasing**
  Different navigation expressions may be same object reference

$$I \models \mathtt{o.age} \doteq 1 \ \texttt{->}\ \langle \mathtt{u.age} = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age} \qquad \textbf{?}$$

  Depends on whether $I \models \mathtt{o} \doteq \mathtt{u}$
- **Method calls**, blocks

# Some Java Features

- Assignments, **complex** expressions with **side effects**
  `int i = 0; if ((i=2) >= 2) {i++;}` // value of i?
- Abrupt termination
  - **Exceptions** (`try` – `catch` – `finally`)
  - **Local jumps** `return`, `break`, `continue`
- **Aliasing**
  Different navigation expressions may be same object reference

  $$I \models \mathtt{o.age} \doteq 1 \;\texttt{->}\; \langle \mathtt{u.age} = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age} \qquad \textbf{?}$$

  Depends on whether $I \models \mathtt{o} \doteq \mathtt{u}$
- **Method calls**, blocks

**Solution within KeY to be discussed in detail**

# More Java Features

**Addressed in KeY**

- Java's rules for localisation of attributes and method implementations (**polymorphism**, **dynamic binding**)

# More Java Features

**Addressed in KeY**

- Java's rules for localisation of attributes and method implementations (**polymorphism**, **dynamic binding**)
    - Scope (class/instance)
    - Context (static/runtime)
    - Visibility
    - `super`

# More Java Features

**Addressed in KeY**

- Java's rules for localisation of attributes and method implementations (**polymorphism**, **dynamic binding**)
  - Scope (class/instance)
  - Context (static/runtime)
  - Visibility
  - `super`

**Solution:** use information from semantic analysis of compiler
(branch proof if implementation not uniquely determined)

# More Java Features

**Addressed in KeY**

- Java's rules for localisation of attributes and method implementations (**polymorphism**, **dynamic binding**)
  - Scope (class/instance)
  - Context (static/runtime)
  - Visibility
  - `super`

  **Solution:** use information from semantic analysis of compiler
  (branch proof if implementation not uniquely determined)

- Run time errors (`null` **pointer exceptions**)
  Functions that model attributes are partially defined

# More Java Features

**Addressed in KeY**

- Java's rules for localisation of attributes and method implementations (**polymorphism**, **dynamic binding**)
  - Scope (class/instance)
  - Context (static/runtime)
  - Visibility
  - `super`

  **Solution:** use information from semantic analysis of compiler
  (branch proof if implementation not uniquely determined)

- Run time errors (`null` **pointer exceptions**)
  Functions that model attributes are partially defined
  **Solution:** optional rule set enforces proof of $\mathtt{!}(o \doteq \mathtt{null})$
  (whenever object reference $o$ accessed)

# More Java Features

- Java Card data types
  `boolean`, `char`, `String`
  `int`, `byte`, `long` (cyclic!)
  Arrays
  **Solution:** optional rule sets $\mathbb{N}$/`int`, rules for built-ins

# More Java Features

- Java Card data types
  `boolean`, `char`, `String`
  `int`, `byte`, `long` (cyclic!)
  Arrays
  **Solution:** optional rule sets $\boldsymbol{N}$/`int`, rules for built-ins
- Java Card **transaction** mechanism (atomic execution)
  "Roll back" uncompleted transactions ("rip out")
  **Solution:** new modality $[\![\alpha]\!]\phi$ "$\phi$ **holds throughout execution of** $\alpha$"

# More Java Features

- Java Card data types
  `boolean`, `char`, `String`
  `int`, `byte`, `long` (cyclic!)
  Arrays
  **Solution:** optional rule sets $\boldsymbol{N}$/`int`, rules for built-ins
- Java Card **transaction** mechanism (atomic execution)
  "Roll back" uncompleted transactions ("rip out")
  **Solution:** new modality $[\![\alpha]\!]\phi$ "$\phi$ **holds throughout execution of** $\alpha$"
- Object **creation** and **initialisation**
  Trick to keep same universe $U$ in all states:
  all objects exist anytime, use attributes $o.\mathrm{created}$, $o.\mathrm{initialized}$

# More Java Features

- Java Card data types
  `boolean`, `char`, `String`
  `int`, `byte`, `long` (cyclic!)
  Arrays
  **Solution:** optional rule sets $\boldsymbol{N}/$`int`, rules for built-ins
- Java Card **transaction** mechanism (atomic execution)
  "Roll back" uncompleted transactions ("rip out")
  **Solution:** new modality $[\![\alpha]\!]\phi$ "$\phi$ **holds throughout execution of** $\alpha$"
- Object **creation** and **initialisation**
  Trick to keep same universe $U$ in all states:
  all objects exist anytime, use attributes $o.\mathrm{created}$, $o.\mathrm{initialized}$
- Formal specification of Java Card API

# Side Effects and Complex Expressions

`int i = 0; if ((i=2) >= 2) {i++;}` // value of i?

JAVA expressions can assign values (**assignment operators**)
**FOL/DL terms** have **no** side effects

# Side Effects and Complex Expressions

```
int i = 0; if ((i=2) >= 2) {i++;} // value of i?
```

JAVA expressions can assign values (**assignment operators**)
**FOL/DL terms** have **no** side effects

**Decomposition** of complex terms following symbolic execution as defined for expressions JAVA language specification
Local **program transformations**

$$\text{ITERATED-ASSIGNMENT}\ \frac{\Gamma\ \texttt{==>}\ \langle\pi\ \mathtt{y} = \mathtt{t};\ \mathtt{x} = \mathtt{y};\omega\rangle\phi,\Delta}{\Gamma\ \texttt{==>}\ \langle\pi\ \mathtt{x} = \mathtt{y} = \mathtt{t};\omega\rangle\phi,\Delta}\qquad t \text{ simple}$$

# Side Effects and Complex Expressions

`int i = 0; if ((i=2) >= 2) {i++;}` // value of i?

JAVA expressions can assign values (**assignment operators**)
**FOL/DL terms** have **no** side effects

**Decomposition** of complex terms following symbolic execution as defined for expressions JAVA language specification
Local **program transformations**

$$\textsc{iterated-assignment}\ \frac{\Gamma \texttt{ ==> } \langle \pi\ \mathtt{y} = \mathtt{t};\ \mathtt{x} = \mathtt{y};\omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \mathtt{x} = \mathtt{y} = \mathtt{t};\omega\rangle\phi, \Delta} \qquad t \text{ simple}$$

Temporary program variables '`_var<n>`' store intermediate results

$$\textsc{if-eval}\ \frac{\Gamma \texttt{ ==> } \langle \pi\ \mathtt{boolean}\ \mathtt{v}_{\mathbf{new}};\ \mathtt{v}_{\mathbf{new}} = \mathtt{b};\ \mathtt{if}\ (\mathtt{v}_{\mathbf{new}})\ \{\alpha\};\ \omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \mathtt{if}\ (b)\ \{\alpha\};\ \omega\rangle\phi, \Delta}$$

where $b$ complex

# Side Effects and Complex Expressions, Cont'd

**Applying rule to statement including guard with side effect is incorrect**

Restrict applicability of IF-THEN and other rules with guards:
Guard expression needs to be **simple** (ie, side effect-free)

$$\textsc{if-split}\ \frac{\Gamma, b \doteq \mathtt{TRUE} \ \mathtt{==>}\ \langle \pi\ \alpha;\ \omega\rangle\phi, \Delta \qquad \Gamma, b \doteq \mathtt{FALSE} \ \mathtt{==>}\ \langle \pi\ \ \omega\rangle\phi, \Delta}{\Gamma \ \mathtt{==>}\ \langle \pi\ \mathtt{if}\ (b)\ \{\alpha\};\ \omega\rangle\phi, \Delta}$$

where $b$ simple

Demo

`javaDL/complex.key`

# Abrupt Termination

Redirection of control flow via `return`, `break`, `continue`, **exceptions**

$$\langle \pi \ \texttt{try} \ \{\xi\alpha\} \ \texttt{catch}(e) \ \{\gamma\} \ \texttt{finally} \ \{\epsilon\}; \ \omega\rangle\phi$$

# Abrupt Termination

Redirection of control flow via `return`, `break`, `continue`, **exceptions**

$$\langle \pi \ \texttt{try}\ \{\xi\alpha\}\ \texttt{catch}(e)\ \{\gamma\}\ \texttt{finally}\ \{\epsilon\};\ \omega\rangle\phi$$

**Solution:** rules work on first active statement, `try` part of prefix

## Abrupt Termination

Redirection of control flow via `return`, `break`, `continue`, **exceptions**

$$\langle \pi \ \texttt{try}\ \{\xi\alpha\}\ \texttt{catch}(e)\ \{\gamma\}\ \texttt{finally}\ \{\epsilon\};\ \omega\rangle\phi$$

**Solution:** rules work on first active statement, `try` part of prefix

TRY-THROW (exc simple)

$$\frac{\Gamma \ \texttt{==>}\ \left\langle \begin{array}{l} \pi\ \texttt{if (exc instanceOf Exception) \{} \\ \quad \texttt{try \{e = exc;}\ \gamma\texttt{\} finally \{}\epsilon\texttt{\}} \\ \texttt{\} else \{}\epsilon\ \texttt{throw exc\};}\ \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \ \texttt{==>}\ \langle \pi\ \texttt{try \{throw exc;}\ \alpha\texttt{\} catch(}e\texttt{) \{}\gamma\texttt{\} finally \{}\epsilon\texttt{\};}\ \omega\rangle\phi, \Delta}$$

# Aliasing

Naive alias resolution causes **proof split** (on $\texttt{o} \doteq \texttt{u}$) at each access

$$\Gamma \texttt{ ==> } \texttt{o.age} \doteq 1 \texttt{ -> } \langle \texttt{u.age} = 2; \rangle \texttt{o.age} \doteq \texttt{u.age}, \Delta$$

# Aliasing

Naive alias resolution causes **proof split** (on $\mathtt{o} \doteq \mathtt{u}$) at each access

$$\Gamma \texttt{ ==> } \mathtt{o.age} \doteq 1 \texttt{ -> } \langle \mathtt{u.age} = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age}, \Delta$$

Unnecessary in many cases!

$$\Gamma \texttt{ ==> } \mathtt{o.age} \doteq 1 \texttt{ -> } \langle \mathtt{u.age} = 2; \quad \mathtt{o.age} = 2; \rangle \mathtt{o.age} \doteq \mathtt{u.age}, \Delta$$

$$\Gamma \texttt{ ==> } \mathtt{o.age} \doteq 1 \texttt{ -> } \langle \mathtt{u.age} = 2; \rangle \mathtt{u.age} \doteq 2, \Delta$$

# Aliasing

Naive alias resolution causes **proof split** (on $\texttt{o} \doteq \texttt{u}$) at each access

$$\Gamma \texttt{ ==> } \texttt{o.age} \doteq 1 \texttt{ -> } \langle \texttt{u.age} = 2; \rangle \texttt{o.age} \doteq \texttt{u.age}, \Delta$$

Unnecessary in many cases!

$$\Gamma \texttt{ ==> } \texttt{o.age} \doteq 1 \texttt{ -> } \langle \texttt{u.age} = 2; \quad \texttt{o.age} = 2; \rangle \texttt{o.age} \doteq \texttt{u.age}, \Delta$$

$$\Gamma \texttt{ ==> } \texttt{o.age} \doteq 1 \texttt{ -> } \langle \texttt{u.age} = 2; \rangle \texttt{u.age} \doteq 2, \Delta$$

**Updates** avoid such proof splits:

- Delayed state computation until clear what **actually** required
- Simplification of updates

# Updates for JAVA

Let *loc* be either one of

- program variable `x`

# Updates for Java

Let *loc* be either one of

- program variable `x`
- attribute access `o.attr` (`o` has object type)

## Updates for JAVA

Let *loc* be either one of

- program variable `x`
- attribute access `o.attr` (`o` has object type)
- array access `a[i]` (`a` has array type, not discussed here)

# Updates for JAVA

Let *loc* be either one of

- program variable `x`
- attribute access `o.attr` (`o` has object type)
- array access `a[i]` (`a` has array type, not discussed here)

$$\text{ASSIGN}\ \frac{\Gamma\ \texttt{==>}\ \{loc := val\}\langle\pi\ \ \omega\rangle\phi, \Delta}{\Gamma\ \texttt{==>}\ \langle\pi\ loc{=}val;\ \omega\rangle\phi, \Delta}$$

where

- *loc* and *val* satisfy above restrictions
- *val* is a side-effect free term,
- $\{loc := val\}$ is DL **update** (usage and semantics as in simple DL)

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \rightsquigarrow t$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{b} \rightsquigarrow (\{\mathtt{o.a} := t\}\mathtt{u}).\mathtt{b}$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{a} \rightsquigarrow$$

```
\if (({o.a:=t}u)=o) \then t \else ({o.a:=t}u).a
```

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \quad \rightsquigarrow \quad t$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{b} \quad \rightsquigarrow \quad (\{\mathtt{o.a} := t\}\mathtt{u}).\mathtt{b}$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{a} \quad \rightsquigarrow$$

```
\if (({o.a:=t}u)=o) \then t \else ({o.a:=t}u).a
```

**Example**

$\{\mathtt{o.a} := \mathtt{o}\}\mathtt{o.a.a.b}$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \rightsquigarrow t$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{b} \rightsquigarrow (\{\mathrm{o.a} := t\}\mathrm{u}).\mathrm{b}$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{a} \rightsquigarrow$$

$$\texttt{\textbackslash if ((\{o.a:=t\}u)=o) \textbackslash then t \textbackslash else (\{o.a:=t\}u).a}$$

**Example**

$$\{\mathrm{o.a} := \mathrm{o}\}\mathrm{o.a.a.b} \rightsquigarrow \{\mathrm{o.a} := \mathrm{o}\}\mathrm{o.a.a.b}$$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \leadsto t$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{b} \leadsto (\{\mathrm{o.a} := t\}\mathrm{u}).\mathrm{b}$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{a} \leadsto$$

$$\texttt{\textbackslash if ((\{o.a:=t\}u)=o) \textbackslash then t \textbackslash else (\{o.a:=t\}u).a}$$

**Example**

$$\{\mathrm{o.a} := \mathrm{o}\}\mathrm{o.a.a.b} \leadsto (\{\mathrm{o.a} := \mathrm{o}\}\mathrm{o.a.a}).\mathrm{b}$$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \rightsquigarrow t$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{b} \rightsquigarrow (\{\mathtt{o.a} := t\}\mathtt{u}).\mathtt{b}$$

$$\{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{a} \rightsquigarrow$$

$$\texttt{\textbackslash if ((\{o.a:=t\}u)=o) \textbackslash then t \textbackslash else (\{o.a:=t\}u).a}$$

**Example**

$$\{\mathtt{o.a} := \mathtt{o}\}\mathtt{o.a.a.b} \rightsquigarrow \begin{array}{l} (\ \texttt{\textbackslash if}\ ((\{\mathtt{o.a} := \mathtt{o}\}\mathtt{o.a}) = \mathtt{o}) \\ \quad \texttt{\textbackslash then o} \\ \quad \texttt{\textbackslash else}\ (\{\mathtt{o.a} := \mathtt{o}\}\mathtt{o.a}).\mathtt{a}\ ).\mathtt{b} \end{array}$$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\begin{array}{lcl} \{\mathbf{o.a} := t\}\mathbf{o.a} & \leadsto & t \\ \{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{b} & \leadsto & (\{\mathtt{o.a} := t\}\mathtt{u}).\mathtt{b} \\ \{\mathtt{o}.\mathbf{a} := t\}\mathtt{u}.\mathbf{a} & \leadsto & \end{array}$$

```
\if (({o.a:=t}u)=o) \then t \else ({o.a:=t}u).a
```

**Example**

$$\{\mathtt{o.a} := \mathtt{o}\}\mathtt{o.a.a.b} \leadsto \begin{array}{l} (\ \texttt{\textbackslash if}\ (\mathtt{o} = \mathtt{o}) \\ \quad \texttt{\textbackslash then}\ \mathtt{o} \\ \quad \texttt{\textbackslash else}\ \mathtt{o.a}\ ).\mathtt{b} \end{array}$$

# Computing Effect of Updates: Attributes

Use **conditional terms** to delay splitting further

$$(\texttt{\textbackslash if (t1 = t2) \textbackslash then t \textbackslash else e})^{I,\beta} = \begin{cases} t^{I,\beta} & t_1^{I,\beta} = t_2^{I,\beta} \\ e^{I,\beta} & \text{otherwise} \end{cases}$$

Computing update followed by **attribute access**

$$\{\mathbf{o.a} := t\}\mathbf{o.a} \rightsquigarrow t$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{b} \rightsquigarrow (\{\mathrm{o.a} := t\}\mathrm{u}).\mathrm{b}$$

$$\{\mathrm{o}.\mathbf{a} := t\}\mathrm{u}.\mathbf{a} \rightsquigarrow$$

```
\if (({o.a:=t}u)=o) \then t \else ({o.a:=t}u).a
```

**Example**

$$\{\mathrm{o.a} := \mathrm{o}\}\mathrm{o.a.a.b} \rightsquigarrow \mathrm{o.b}$$

# Parallel Updates

Computing update followed by **update**

$$\{l_1 := r_1\}\{l_2 := r_2\} \quad = \quad \{\{l_1 := r_1\},\ \{\{l_1 := r_1\} \downarrow l_2 := \{l_1 := r_1\}r_2\}\}$$

Results in **parallel update**

# Parallel Updates

Computing update followed by **update**

$$\{l_1 := r_1\}\{l_2 := r_2\} \quad = \quad \{\{l_1 := r_1\},\ \{\{l_1 := r_1\} \downarrow l_2 := \{l_1 := r_1\}r_2\}\}$$

Results in **parallel update**

**Syntax**

$$\{l_1 := v_1,\ \ldots,\ l_n := v_n\}$$

# Parallel Updates

Computing update followed by **update**

$$\{l_1 := r_1\}\{l_2 := r_2\} \quad = \quad \{\{l_1 := r_1\},\ \{\{l_1 := r_1\} \downarrow l_2 := \{l_1 := r_1\} r_2\}\}$$

Results in **parallel update**

**Syntax**

$$\{l_1 := v_1,\ \ldots,\ l_n := v_n\}$$

**Semantics**

- All $l_i$ and $v_i$ computed in old state
- All updates done simultaneously
- If conflict $\quad l_i = l_j,\ v_i \neq v_j \quad$ later update wins

# Method Call

**Method call** with actual parameters $arg_1, \ldots, arg_n$

$$\{arg_1 := t_1, \ldots, arg_n := t_n,\ o := t_o\}\langle o.m(arg_1, \ldots, arg_n);\rangle\phi$$

Where method declaration is: `void` $m(T_1\ p_1, \ldots, T_n\ p_n)$

# Method Call

**Method call** with actual parameters $arg_1, \ldots, arg_n$

$$\{arg_1 := t_1, \ldots, arg_n := t_n,\ o := t_o\}\langle o.m(arg_1, \ldots, arg_n); \rangle\phi$$

Where method declaration is: `void` $m(T_1\ p_1, \ldots, T_n\ p_n)$

What the rule **method-call** does:

- (type conformance of $arg_i$ to $T_i$ guaranteed by JAVA compiler)

# Method Call

**Method call** with actual parameters $arg_1, \ldots, arg_n$

$$\{arg_1 := t_1, \ldots, arg_n := t_n,\ o := t_o\}\langle o.m(arg_1, \ldots, arg_n);\rangle\phi$$

Where method declaration is: `void` $m(T_1\ p_1, \ldots, T_n\ p_n)$

What the rule **method-call** does:

- (type conformance of $arg_i$ to $T_i$ guaranteed by JAVA compiler)
- for each formal parameter $p_i$ of $m$ declare & initialize new local variable '$T_i\ p_i = arg_i$;'

# Method Call

**Method call** with actual parameters $arg_1, \ldots, arg_n$

$$\{arg_1 := t_1, \ldots, arg_n := t_n,\ o := t_o\}\langle o.m(arg_1, \ldots, arg_n); \rangle\phi$$

Where method declaration is: `void` $m(T_1\ p_1, \ldots, T_n\ p_n)$

What the rule **method-call** does:

- (type conformance of $arg_i$ to $T_i$ guaranteed by JAVA compiler)
- for each formal parameter $p_i$ of $m$ declare & initialize new local variable '$T_i\ p_i = arg_i$;'
- look up implementation class $C$ of $m$ split proof, if implementation not determinable

# Method Call

**Method call** with actual parameters $arg_1, \ldots, arg_n$

$$\{arg_1 := t_1, \ldots, arg_n := t_n,\ o := t_o\}\langle o.m(arg_1, \ldots, arg_n);\rangle\phi$$

Where method declaration is: `void` $m(T_1\ p_1, \ldots, T_n\ p_n)$

What the rule **method-call** does:

- (type conformance of $arg_i$ to $T_i$ guaranteed by JAVA compiler)
- for each formal parameter $p_i$ of $m$ declare & initialize new local variable '$T_i\ p_i = arg_i$;'
- look up implementation class $C$ of $m$ split proof, if implementation not determinable
- make concrete call `o`.$C$::$m(p_1, \ldots, p_n)$

# Method Body Expand

After processing code that binds actual to formal parameters (symbolic execution of '$T_i\ p_i = arg_i$;')

$$\text{METHOD-BODY-EXPAND}\ \frac{\Gamma \texttt{ ==> } \langle \pi\ \texttt{method-frame}(C(\texttt{o}))\texttt{\{}\ b\ \texttt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \texttt{o}.C{::}m(p_1, \ldots, p_n);\ \omega\rangle\phi, \Delta}$$

# Method Body Expand

After processing code that binds actual to formal parameters (symbolic execution of '$T_i\ p_i = arg_i;$')

$$\text{METHOD-BODY-EXPAND}\ \frac{\Gamma\ \texttt{==>}\ \langle \pi\ \texttt{method-frame}(C(\texttt{o}))\texttt{\{}\ b\ \texttt{\}}\ \omega\rangle\phi, \Delta}{\Gamma\ \texttt{==>}\ \langle \pi\ \texttt{o}.C{::}m(p_1, \ldots, p_n);\ \omega\rangle\phi, \Delta}$$

Symbolic Execution

# Method Body Expand

After processing code that binds actual to formal parameters (symbolic execution of '$T_i\ p_i = arg_i$;')

$$\textsc{method-body-expand}\ \frac{\Gamma \texttt{ ==> } \langle \pi\ \texttt{method-frame}(C(\texttt{o}))\texttt{\{}\ b\ \texttt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \texttt{o}.C{::}m(p_1, \ldots, p_n);\ \omega\rangle\phi, \Delta}$$

Symbolic Execution
Only static information available, proof splitting

# Method Body Expand

After processing code that binds actual to formal parameters (symbolic execution of '$T_i\ p_i = arg_i$;')

$$\text{METHOD-BODY-EXPAND}\ \frac{\Gamma \texttt{ ==> } \langle \pi\ \texttt{method-frame}(C(\texttt{o}))\texttt{\{}\ b\ \texttt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \texttt{ ==> } \langle \pi\ \texttt{o}.C{::}m(p_1, \ldots, p_n);\ \omega\rangle\phi, \Delta}$$

Symbolic Execution
Only static information available, proof splitting
Runtime infrastructure required in calculus