Specification & Formal Analysis of Java Programs
Functional Verification of Java Programs

Prof. Dr. Bernhard Beckert | ADAPT 2010
Java Type Hierarchy

Signature based on Java’s type hierarchy

Each class referenced in API and target program is in signature with appropriate partial order
Modelling Attributes in FOL

Modeling instance attributes

- Each \( o \in D_{\text{Person}} \) has associated \( \text{age} \) value
- \( I(\text{age}) \) is function from \( \text{Person} \) to \( \text{int} \)
- Attribute values can be changed
- For each class \( C \) with attribute \( a \) of type \( T \):
  - FSym\(_{nr}\) declares non-rigid function \( T \ a(C) \);

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>int age</td>
</tr>
<tr>
<td>int id</td>
</tr>
<tr>
<td>int setAge(int i)</td>
</tr>
<tr>
<td>int getId()</td>
</tr>
</tbody>
</table>

Attribute Access

Signature FSym\(_{nr}\): \( \text{int age(Person)}; \)  \( \text{Person p;} \)

Java/JML expression \( p.\text{age} >= 0 \)
Typed FOL \( \text{age(p)} >= 0 \)
Modelling Attributes in FOL

Modeling instance attributes

- Each $o \in D_{Person}$ has associated $age$ value
- $I(age)$ is function from $Person$ to $int$
- Attribute values can be changed
- For each class $C$ with attribute $a$ of type $T$: $FSym_{nr}$ declares non-rigid function $T\ a(C)$;

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>int age</td>
</tr>
<tr>
<td>int id</td>
</tr>
</tbody>
</table>

- $\text{int setAge(int i)}$
- $\text{int getId()}$

Attribute Access

Signature $FSym_{nr}$: $\text{int age(Person);}$ $\text{Person p;}$

Java/JML expression $\text{p.age} >= 0$

Typed FOL $\text{age(p)} >= 0$
Modelling Attributes in FOL

Modeling instance attributes

- Each \( o \in D^{\text{Person}} \) has associated \textit{age} value
- \( I(\text{age}) \) is \textit{function} from \textit{Person} to \textit{int}
- Attribute values can be changed
- For each class \( C \) with attribute \( a \) of type \( T \): 
  \[ F\text{Sym}_{nr} \text{ declares } \textit{non-rigid function } T \ a(C); \]

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{int} age</td>
</tr>
<tr>
<td>\textbf{int} id</td>
</tr>
<tr>
<td>\textbf{int} setAge(int \ i)</td>
</tr>
<tr>
<td>\textbf{int} getId()</td>
</tr>
</tbody>
</table>

Attribute Access

Signature \( F\text{Sym}_{nr} : \quad \textbf{int} \ \text{age(Person)}; \quad \text{Person } p; \)

Java/JML expression \( p.\text{age} >= 0 \)

Typed FOL \( \text{age}(p) >= 0 \)
Modelling Attributes in FOL

Modeling instance attributes

- Each \( o \in D_{\text{Person}} \) has associated \( \text{age} \) value
- \( I(\text{age}) \) is function from \( \text{Person} \) to \( \text{int} \)
- Attribute values can be changed
- For each class \( C \) with attribute \( a \) of type \( T \):
  \( \text{FSym}_{nr} \) declares "non-rigid" function
  \( T \ a(C) \);

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>int age</td>
</tr>
<tr>
<td>int id</td>
</tr>
<tr>
<td>int setAge(int i)</td>
</tr>
<tr>
<td>int getId()</td>
</tr>
</tbody>
</table>

Attribute Access

Signature \( \text{FSym}_{nr} \): \( \text{int age(Person)}; \quad \text{Person p;} \)

Java/JML expression \( p.age \geq 0 \)

Typed FOL \( \text{age(p)} \geq 0 \)
Modelling Attributes in FOL

Modeling instance attributes

- Each \( o \in D^{\text{Person}} \) has associated \( \text{age} \) value
- \( I(\text{age}) \) is function from \( \text{Person} \) to \( \text{int} \)
- Attribute values can be changed
- For each class \( C \) with attribute \( a \) of type \( T \):
  
  \( \text{FSym}_{nr} \) declares \textit{non-rigid} function \( T_{a(C)} \);

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ( \text{age} )</td>
</tr>
<tr>
<td>int ( \text{id} )</td>
</tr>
<tr>
<td>int ( \text{setAge} ) (int ( i ))</td>
</tr>
<tr>
<td>int ( \text{getId} () )</td>
</tr>
</tbody>
</table>

Attribute Access

Signature \( \text{FSym}_{nr} : \) \textbf{int} \( \text{age}(\text{Person}); \) \textbf{Person} \( p; \)

Java/JML expression \( p.\text{age} \geq 0 \)

Typed FOL \( \text{age}(p) \geq 0 \)
Properties of attributes

- When not initialized, $I(a) = \text{null}$
- Overloading can be resolved by qualifying with class path:
  $\text{Person::p.age}$

Changing the value of attributes

How to translate assignment to attribute $p\text{.age}=17; \ ?$

$$\text{assign } \frac{\Gamma \Rightarrow \{l := t\}\langle\text{rest}\rangle\phi, \Delta}{\Gamma \Rightarrow \langle l = t; \text{ rest}\rangle\phi, \Delta}$$

Admit on left-hand side of update *program location expressions*
Properties of attributes

- When not initialized, \( T(a) = \text{null} \)
- Overloading can be resolved by qualifying with class path: \text{Person::p.age} \n
Changing the value of attributes

How to translate assignment to attribute \( p.age = 17; \) ?

\[
\text{assign} \quad \Gamma \implies \{ p.age := 17 \} \langle \text{rest} \rangle \phi, \Delta \\
\Gamma \implies \langle p.age = 17; \text{rest} \rangle \phi, \Delta
\]

Admit on left-hand side of update program location expressions
A Warning

Computing the effect of updates with attribute locations is complex

Example

- Signature $\text{FSym}_{nr}: \quad \text{C } a (\text{C}) ; \quad \text{C } b (\text{C}) ; \quad \text{C } o ;$

- Consider $\{ \text{o.a.a := o} \}{ \{ \text{o.b.a := o.a} \}$

- First update may affect of second update

- $\text{o.a}$ and $\text{o.b}$ might refer to same object (be aliases)

KeY applies rules automatically, you don’t need to worry about details
A Warning

Computing the effect of updates with attribute locations is complex

Example

- Signature $\text{FSym}_{nr}$: $\text{C a(C)}$; $\text{C b(C)}$; $\text{C o;}$
- Consider $\{\text{o.a.a := o}\}{\text{o.b.a := o.a}}$
- First update may affect of second update
- $\text{o.a}$ and $\text{o.b}$ might refer to same object (be aliases)

KeY applies rules automatically, you don’t need to worry about details
A Warning

Computing the effect of updates with attribute locations is complex

Example

- Signature FSym\textsubscript{nr}: C a(C); C b(C); C o;
- Consider \{o.a.a := o\}\{o.b.a := o.a\}
- First update may affect of second update
- o.a and o.b might refer to same object (be aliases)

KeY applies rules automatically, you don’t need to worry about details
A Warning

Computing the effect of updates with attribute locations is complex

Example

- Signature $FSym_{nr}$: $\text{C a (C)}; \text{C b (C)}; \text{C o} ;$
- Consider $\{\text{o.a.a := o}\}{\text{o.b.a := o.a}}$
- First update may affect of second update
- $\text{o.a}$ and $\text{o.b}$ might refer to same object (be aliases)

KeY applies rules automatically, you don’t need to worry about details
A Warning

Computing the effect of updates with attribute locations is complex

Example

- Signature $\text{FSym}_{nr}$: $\text{C a(C)}$; $\text{C b(C)}$; $\text{C o}$;
- Consider $\{\text{o.a.a := o}\}\{\text{o.b.a := o.a}\}$
- First update may affect of second update
- $\text{o.a}$ and $\text{o.b}$ might refer to same object (be aliases)

KeY applies rules automatically, you don’t need to worry about details
## Modeling class (static) attributes

For each class $C$ with static attribute $a$ of type $T$:

- $FSym_{nr}$ declares *non-rigid* constant $T \ a$;

- Value of $a$ is $I(a)$ for all instances of $C$

- If necessary, qualify with class (path):
  - `byte java.lang.Byte.MAX_VALUE`

- Standard values are predefined in KeY:
  - $I(\text{byte java.lang.Byte.MAX_VALUE}) = 127$
Modeling reference \textit{this} to the \textit{receiving object}

Special name for the object whose Java code is currently executed:

- in JML: \texttt{Object self;}
- in Java: \texttt{Object this;}
- in KeY: \texttt{Object self;}

Default assumption in JML-KeY translation: \texttt{!(self = null)}
Which Objects do Exist?

How to model object creation with new?

**Constant Domain Assumption**

Assume that domain $D$ is the same in all states of LTS $K = (S, \rho)$

*Desirable consequence:*

Validity of *rigid* FOL formulas unaffected by programs

$$\models \forall T x; \phi \rightarrow [p](\forall T x; \phi)$$

is valid for rigid $\phi$

**Realizing Constant Domain Assumption**

- Non-rigid function $\text{boolean} \ <\text{created}>\ (\text{Object});$
- Equal to $\text{true}$ iff argument object has been created
- Initialized as $I(<\text{created}>)(o) = F$ for all $o \in D$
- Object creation modeled as $\{o.\langle \text{created} \rangle := \text{true} \}$ for next "free" $o$
Which Objects do Exist?

How to model *object creation* with **new**?

**Constant Domain Assumption**

Assume that domain $\mathcal{D}$ is the same in all states of LTS $K = (S, \rho)$

*Desirable consequence:*
Validity of *rigid* FOL formulas unaffected by programs

$$\models \forall T \, x; \, \phi \rightarrow [p](\forall T \, x; \, \phi) \quad \text{is valid for rigid } \phi$$

**Realizing Constant Domain Assumption**

- Non-rigid function `boolean <created>(Object);`
- Equal to `true` iff argument object has been created
- Initialized as $I(<created>)(o) = F$ for all $o \in \mathcal{D}$
- Object creation modeled as `{o.<created> := true}` for
Which Objects do Exist?

How to model object creation with `new`?

**Constant Domain Assumption**

Assume that domain $\mathcal{D}$ is the same in all states of LTS $K = (S, \rho)$

*Desirable consequence:*

Validity of *rigid* FOL formulas unaffected by programs

$$\models \forall T \ x; \ \phi \rightarrow [p](\forall T \ x; \ \phi) \quad \text{is valid for rigid } \phi$$

**Realizing Constant Domain Assumption**

- Non-rigid function `boolean <created>(Object);`
- Equal to `true` iff argument object has been created
- Initialized as $I(<created>)(o) = F$ for all $o \in \mathcal{D}$
- Object creation modeled as $\{o.<created> := \text{true}\}$ for
Quantified Updates

Initialization of all objects in a given class $C$

- Specify that default value of attribute $\text{int } a(C)$ is 0
- Can use $\forall C.o; o.a \neq 0$ in premise
- Problem: difficult to exploit for update simplification

Definition (Quantified Update)

For $T$ well-ordered type (no $\infty$ descending chains): quantified update:

$$\{ \text{\texttt{for T x; if P; l := r}} \}$$

- For all objects $d$ in $D^T$ such that $\beta^d_x \models P$
  perform the updates $\{ l := r \}$ under $\beta^d_x$ in parallel
- If there are several $l$ with conflicting $d$ then choose
Quantified Updates

Initialization of all objects in a given class \( C \)

- Specify that default value of attribute \( \text{int } a(C) \) is 0
- Can use \( \forall C \ o; o.a \neq 0 \) in premise

| \( C \) | \( \text{int } a \) |

Problem: difficult to exploit for update simplification

Definition (Quantified Update)

For \( T \) well-ordered type (no \( \infty \) descending chains): quantified update:

\[
\{ \textbf{\text{\textbackslash for } } T \ x; \ \textbf{\textbackslash if } P; l := r \} \]

- For all objects \( d \) in \( D^T \) such that \( \beta_x^d \models P \)
  perform the updates \( \{ l := r \} \) under \( \beta_x^d \) in parallel
- If there are several \( l \) with conflicting \( d \) then choose
Quantified Updates

Initialization of all objects in a given class \( C \)

- Specify that default value of attribute \( \text{int} \ a(C) \) is 0
- Can use \( \forall C \ o; \ o.a = 0 \) in premise
- \textbf{Problem}: difficult to exploit for update simplification

\[
\begin{array}{|c|c|}
\hline
\text{C} & \text{int} \ a \\
\hline
\end{array}
\]

Definition (Quantified Update)

For \( T \) well-ordered type (no \( \infty \) descending chains): \textit{quantified update}:

\[
\{ \text{\texttt{for T x; \ if P; l := r}} \}
\]

- For all objects \( d \) in \( D^T \) such that \( \beta_x^d \models P \)
- perform the updates \( \{ l := r \} \) under \( \beta_x^d \) in \textit{parallel}
- If there are several \( l \) with conflicting \( d \) then choose...
Quantified Updates

Initialization of all objects in a given class $C$

- Specify that default value of attribute `int a(C)` is 0
- Can use $\forall C \ o; \ o.a \neq 0$ in premise
- *Problem:* difficult to exploit for update simplification

Definition (Quantified Update)

For $T$ well-ordered type (no $\infty$ descending chains): *quantified update:*

```
{ \text{for } T \ x; \ \text{if } P; \ l := r}
```

- For all objects $d$ in $D^T$ such that $\beta_x^d \models P$
  perform the updates $\{l := r\}$ under $\beta_x^d$ in *parallel*
- If there are several $l$ with conflicting $d$ then choose
Quantified Updates Cont’d

- The conditional expression is optional
- Typically, \( x \) occurs in \( \text{P} \), \( \text{l} \), and \( \text{r} \) (but doesn’t need to)
- There is a *normal form* for updates computed efficiently by KeY

Example (Integer types are well-ordered in KeY—Demo)

\[ \exists \text{int } n; (\{\text{for int } i; \ l := i\}(l = n)) \]

- Is valid both for Java \text{int} and \( \mathbb{Z} \) (\( n \div 0 \) non-standard order)
- Proven automatically by update simplifier

Example (Initialization of field \( a \) for all objects in class \( C \))

\[ \{\text{for } T \ o; \ o.a := 0\} \]
The conditional expression is optional

Typically, $x$ occurs in $P$, $l$, and $r$ (but doesn’t need to)

There is a *normal form* for updates computed efficiently by KeY

---

**Example (Integer types are well-ordered in KeY—Demo)**

\[ \exists \text{int } n; (\{\text{for int } i; l := i\} (l = n)) \]

- Is valid both for Java `int` and $\mathbb{Z}$ ($n \equiv 0$ non-standard order)
- Proven automatically by update simplifier

---

**Example (Initialization of field $a$ for all objects in class $C$)**

\[ \{\text{for } T o; o.a := 0\} \]
Quantified Updates Cont’d

- The conditional expression is optional
- Typically, $x$ occurs in $P$, $l$, and $r$ (but doesn’t need to)
- There is a normal form for updates computed efficiently by KeY

Example (Integer types are well-ordered in KeY—Demo)

$$\exists \text{int } n; \left( \{ \text{for int } i; \ l := i \} (l = n) \right)$$
- Is valid both for Java int and $\mathbb{Z}$ ($n \geq 0$ non-standard order)
- Proven automatically by update simplifier

Example (Initialization of field $a$ for all objects in class $C$)

$$\{ \text{for } T \ o; \ o.a := 0 \}$$
Extending Dynamic Logic to Java

Any syntactically correct Java with some extensions

- Needs not be compilable unit
- Permit externally declared, non-initialized variables
- Referenced class definitions loaded in background

And some limitations . . .

- No concurrency
- No generics
- No Strings
- No I/O
- No floats
- No dynamic class loading or reflexion
- API method calls: need either JML contract or implementation
Arrays

Java type hierarchy includes array types

- T
- Object
- Object[
- Object[][]
- ...
- ⊥

Arrays a and b can refer to same object (aliases)

Model array with non-rigid function T[](C, int)

Instead of [i](ar)
write ar[i]

Value of entry in array T[] ar; defined in class C depends on reference ar to array in C and index i.
Java Features in Dynamic Logic: Arrays

Java type hierarchy includes array types

Arrays

- $T$
- $\triangleright$
- $\text{Object}$
- $\triangleright$
- $\text{Object[]}$
- $\triangleright$
- $\text{Object[][]}$
- $\text{...}$
- $\bot$

Non-rigid functions modeling attributes can have array type. Value of entry in array $T[a_i]$ defined in class $C$ depends on reference $ar$ to array in $C$ and index $i$

Model array with non-rigid function $T[a_i](C,i)$

Instead of $[a_i]$ write $ar[i]$

Arrays $a$ and $b$ can refer to same object (aliases)

KeY implements update application and simplification rules for array locations
Java Features in Dynamic Logic: Arrays

Arrays

Java type hierarchy includes array types

Object[]

Object

T

⊥
Java Features in Dynamic Logic: Arrays

Java type hierarchy includes array types

Arrays

T

Object

Object[]

Object[][]

...
Java Features in Dynamic Logic: Arrays

Java type hierarchy includes array types

- Arrays
- T
- Object
- Object[]
- Object[][]
- ⊥
Java Features in Dynamic Logic: Arrays

Java type hierarchy includes array types

Arrays

\[ T \]

\[ \triangleleft \]

Object

\[ \triangleleft \]

Object[]

\[ \triangleleft \]

Object[][]

...
Java Features in Dynamic Logic: Arrays

Java type hierarchy includes array types

Arrays

Object

Object[]

Object[][]

...
Java Features in Dynamic Logic:
Arrays

Java type hierarchy includes array types

Arrays

\[
\begin{align*}
T & \downarrow \\
\text{Object} & \downarrow \\
\text{Object[]} & \downarrow \\
\text{Object[]}[] & \downarrow \\
\bot & 
\end{align*}
\]
Java Features in Dynamic Logic: Complex Expressions

Complex expressions with side effects
- Java expressions may contain assignment operator with side effect
- FOL terms have no side effect on the state
- Java expressions can be complex and nested

Example (Complex expression with side effects in Java)

```java
int i = 0; if ((i=2) >= 2) i++; value of i ?
```
Decomposition of complex terms by symbolic execution

Follow the rules laid down in Java Language Specification

Local code transformations

\[
\text{evalOrderIteratedAssignmt} \quad \frac{\Gamma \Rightarrow \langle y = t; x = y; \text{rest} \rangle \phi, \Delta}{\Gamma \Rightarrow \langle x = y = t; \text{rest} \rangle \phi, \Delta}
\]

Temporary variables store result of evaluating subexpression

\[
\text{ifEval} \quad \frac{\Gamma \Rightarrow \langle \text{boolean } v0; v0 = b; \text{if } (v0) p; r \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \text{if } (b) p; r \rangle \phi, \Delta}
\]

Guards of conditionals/loops always evaluated (hence: side effect-free)

before conditional/unwind rules applied
Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

\[ \langle \pi \text{ try } \xi \text{ p catch (e) q finally r; } \omega \rangle \phi \]

Rules ignore inactive prefix, work on active statement, leave postfix

Rule tryThrow matches try–catch in pre-/postfix and active throw

\[ \Rightarrow \langle \pi \text{ if (e instanceof T) \{try } x=e; q \text{ finally r} \} \text{ else } \{ r; \text{ throw e; } p \} \rangle \]

\[ \Rightarrow \langle \pi \text{ try } \{ \text{ throw e; } p \} \text{ catch (T x) q finally r; } \omega \rangle \]
Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

\[ \langle \pi \text{try} \xi p \text{catch}(e) q \text{finally} r; \omega \rangle \phi \]

Rules ignore inactive prefix, work on active statement, leave postfix

Rule tryThrow matches try–catch in pre-/postfix and active throw

\[ \Rightarrow \langle \pi \text{if} (e \text{ instanceof } T) \{ \text{try} x=e; q \text{finally} r \} \text{else} \{ r; \text{throw} e; p \} \text{catch}(T x) q \text{finally} r; \omega \rangle \phi \]
Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

\[ \langle \pi \text{ try } \xi \text{ catch } (e) \text{ q finally } r; \omega \rangle \phi \]

Rules ignore inactive prefix, work on active statement, leave postfix

Rule tryThrow matches try–catch in pre-/postfix and active throw

\[ \Rightarrow \langle \pi \text{ if } (e \text{ instanceof } T) \{ \text{ try } x=e; \text{ q finally } r \} \text{ else } \{ r; \text{ throw } e \} \rangle \]

\[ \Rightarrow \langle \pi \text{ try } \{ \text{ throw } e; \text{ p} \} \text{ catch } (T x) \text{ q finally } r; \omega \rangle \]
Java Features in Dynamic Logic: Aliasing

Reference Aliasing

Naive alias resolution causes *proof split* (on $o \equiv u$) at each access

$$\Rightarrow o.\text{age} \equiv 1 \rightarrow \langle u.\text{age} = 2; \rangle o.\text{age} \equiv u.\text{age}$$

Unnecessary case analyses

$$\Rightarrow o.\text{age} \equiv 1 \rightarrow \langle u.\text{age} = 2; o.\text{age} = 2; \rangle o.\text{age} \equiv u.\text{age}$$

$$\Rightarrow o.\text{age} \equiv 1 \rightarrow \langle u.\text{age} = 2; \rangle u.\text{age} \equiv 2$$

Updates avoid case analyses—Demo

`lect13/alias2.key`
Java Features in Dynamic Logic: Aliasing

Reference Aliasing

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

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; \rangle o.\text{age} \doteq u.\text{age}$$

Unnecessary case analyses

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; o.\text{age} = 2; \rangle o.\text{age} \doteq u.\text{age}$$

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; \rangle u.\text{age} \doteq 2$$

Updates avoid case analyses— Demo

lect13/alias2.key
## Reference Aliasing

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

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; \rangle o.\text{age} \doteq u.\text{age}$$

## Unnecessary case analyses

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; \rangle o.\text{age} \doteq u.\text{age}$$

$$\Rightarrow o.\text{age} \doteq 1 \rightarrow \langle u.\text{age} = 2; \rangle u.\text{age} \doteq 2$$

## Updates avoid case analyses— Demo

lect13/alias2.key
Aliasing Cont’d

Form of Java program locations

- Program variable $x$
- Attribute access $o.a$
- Array access $ar[i]$

Assignment rule for arbitrary Java locations

$$\begin{align*}
\text{assign} & \quad \Gamma \Rightarrow U\{1 := t\}\langle \pi \omega \rangle \phi, \Delta \\
& \quad \Gamma \Rightarrow U\langle \pi \quad 1 = t; \quad \omega \rangle \phi, \Delta
\end{align*}$$

Updates in front of program formula (= current state) carried over

- Rules for applying updates complex for reference types
- Aliasing analysis causes case split: delayed using conditional terms

\[
\{ o.a := t \} \quad u.a \quad \text{if} \quad \{ o.a := t \} \quad u.a \quad \text{else}
\]
Java Features in Dynamic Logic: Method Calls

**Method Call with actual parameters** \( arg_0, \ldots, arg_n \)

\[
\{ \text{arg}_0 := t_0 \parallel \cdots \parallel \text{arg}_n := t_n \parallel c := t_c \} \langle c.\text{m}(\text{arg}_0, \ldots, \text{arg}_n) ; \rangle \phi
\]

where \( m \) declared as \texttt{void} \( m(T_0 \ p_0, \ldots, T_n \ p_n) \)

**Actions of rule** \texttt{methodCall}:

- (type conformance of \( \text{arg}_i \) to \( T_i \) guaranteed by Java compiler)
- for each formal parameter \( p_i \) of \( m \):
  - declare & initialize new local variable \( T_i \ p \# i = \text{arg}_i \);
- look up implementation class \( C \) of \( m \) and split proof if implementation cannot be uniquely determined
- create method invocation \( c.\text{m}(p \# 0, \ldots, p \# n)@C \)
Method Body Expand

1. Execute code that binds actual to formal parameters
   \[ T_i \ p#i = arg_i; \]

2. Call rule \textit{methodBodyExpand}

   \[
   \Gamma \Rightarrow \langle \pi \ \text{method-frame}(source=C, \ this=c) \{ \ \text{body} \} \ \omega \rangle \phi, \Delta
   \]

   \[
   \Gamma \Rightarrow \langle \pi \ c.m(p#0, \ldots, p#n) @C; \ \omega \rangle \phi, \Delta
   \]
Method Calls Cont’d

Method Body Expand

1. Execute code that binds actual to formal parameters
   \[ T_i \ p#i = arg_i; \]

2. Call rule \textit{methodBodyExpand}

\[ \Gamma \Rightarrow \langle \pi \ \text{method-frame(source=C, this=c)} \{ \text{body} \} \ \omega \rangle \phi, \Delta \]
\[ \Gamma \Rightarrow \langle \pi \ c.m(p#0, \ldots, p#n) @C; \ \omega \rangle \phi, \Delta \]

Symbolic Execution
Only static information available, proof splitting

Demo

lect13/method2.key
Method Body Expand

1. Execute code that binds actual to formal parameters
   \[ T_i \, p#i = arg_i; \]

2. Call rule methodBodyExpand

\[ \Gamma \Rightarrow \langle \pi \text{method-frame}(source=C, \text{this}=c) \{ \text{body} \} \omega \rangle \phi, \Delta \]

Symbolic Execution
Runtime infrastructure required in calculus

Demo

lect13/method2.key
Method Body Expand

1. Execute code that binds actual to formal parameters
   \[ T_i \ p^#_i = \text{arg}_i; \]

2. Call rule \textit{methodBodyExpand}

\[
\Gamma \Rightarrow \langle \pi \text{ method-frame(source=C, this=c)} \{ \text{body} \} \omega \rangle \phi, \Delta
\]

Symbolic Execution
Runtime infrastructure required in calculus

Demo

\texttt{lect13/method2.key}
Localisation of Fields and Method Implementation

Java has complex rules for *localisation* of attributes and method implementations

- Polymorphism
- Late binding
- Scoping (class vs. instance)
- Context (static vs. runtime)
- Visibility (private, protected, public)

*Use information from semantic analysis of compiler framework*

Proof split into cases when implementation not statically determined
Null pointer exceptions

There are no “exceptions” in FOL: $\mathcal{I}$ total on FSym

Need to model possibility that $o \models \text{null}$ in $o.a$

- KeY creates PO for $!o \models \text{null}$ upon each field access
- Can be switched off with option $nullPointerPolicy$
Object initialization

Java has complex rules for object initialization

- Chain of constructor calls until `Object`
- Implicit calls to `super()`
- Visibility issues
- Initialization sequence

Coding of initialization rules in methods `<createObject>()`, `<init>()`, ...
which are then symbolically executed
Formal specification of Java API

How to perform symbolic execution when Java API method is called?

1. API method has reference implementation in Java
   - Call method and execute symbolically
   - Problem: Reference implementation not always available
   - Problem: Too expensive

2. Use JML contract of API method:
   1. Show that requires clause is satisfied
   2. Obtain postcondition from ensures clause
   3. Delete updates with modifiable locations from symbolic state

Java Card API in JML or DL

DL version available in KeY, JML work in progress. See W. Mostowski.

www.cs.ru.nl/~woj/software/software.html
A Round Tour of Java Features in DL
Cont’d

Formal specification of Java API

How to perform symbolic execution when Java API method is called?

1. API method has reference implementation in Java
   Call method and execute symbolically
   Problem Reference implementation not always available
   Problem Too expensive

2. Use JML contract of API method:
   1. Show that requires clause is satisfied
   2. Obtain postcondition from ensures clause
   3. Delete updates with modifiable locations from symbolic state

Java Card API in JML or DL

DL version available in KeY, JML work in progress
See W.
Summary

- Most Java features covered in KeY
- Several of remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is high
- Proving loops requires user to provide invariant
  - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic
### Literature for this Lecture

**Essential**

| KeY Book | Verification of Object-Oriented Software (see course web page), Chapter 3: *Dynamic Logic*, Sections 3.6.1, 3.6.2, 3.6.5, 3.6.7 |

**Recommended**

| KeY Book | Verification of Object-Oriented Software (see course web page), Chapter 3: *Dynamic Logic*, Section 3.9 |