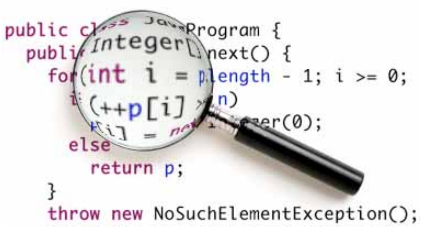
# 15-819M: Data, Code, Decisions

## 08: Reasoning about Java Programs with Dynamic Logic

André Platzer

aplatzer@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA



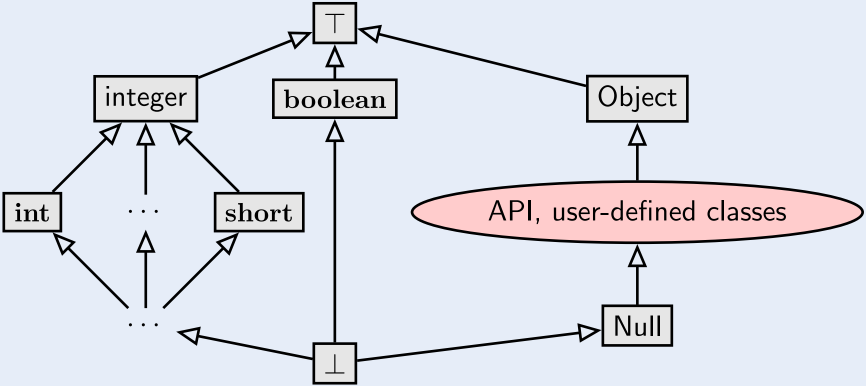
# Outline

# Outline

1. Java DL
   - Java Type Hierarchy
   - Modeling OO Programs
   - Self
   - Object Creation
2. Quantified Updates
3. Round Tour
   - Java Programs
   - Arrays
   - Side Effects
   - Abrupt Termination
   - Aliasing
   - Method Calls
   - Null Pointers
   - Initialization
   - API
   - API
4. Summary

# 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

| Person |
| --- |
| **int** age<br>**int** id |
| **int** setAge(**int** i)<br>**int** getId() |

- Each $o \in \mathtt{D}^{\mathsf{Person}}$ has associated age value

# Modelling Attributes in FOL

## Modeling instance attributes

| Person |
| --- |
| **int** age<br>**int** id |
| **int** setAge(**int** i)<br>**int** getId() |

- Each $o \in \mathtt{D}^{\mathsf{Person}}$ has associated age value
- $\mathcal{I}$(age) is function from Person to **int**

# Modelling Attributes in FOL

## Modeling instance attributes

| Person |
| --- |
| **int** age<br>**int** id |
| **int** setAge(**int** i)<br>**int** getId() |

- Each $o \in \mathtt{D}^{\text{Person}}$ has associated age value
- $\mathcal{I}(\text{age})$ is function from Person to **int**
- Attribute values can be changed

# Modelling Attributes in FOL

## Modeling instance attributes

```
Person
-----------------------
int age
int id
-----------------------
int setAge(int i)
int getId()
```

- Each $o \in \mathtt{D}^{\mathsf{Person}}$ has associated age value
- $\mathcal{I}(\mathsf{age})$ is function from Person to **int**
- Attribute values can be changed
- For each class $C$ with attribute a of type $T$: $\mathsf{FSym}_{nr}$ declares flexible function $T$ a$(C)$;

# Modelling Attributes in FOL

## Modeling instance attributes

| Person |
| --- |
| int age<br>int id |
| int setAge(int i)<br>int getId() |

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

## Attribute Access

Signature $\mathrm{FSym}_{nr}$: **int** age(Person); Person p;

JAVA/JML expression `p.age >= 0`

Typed FOL $\mathrm{age}(\mathrm{p}) >= 0$

KeY postfix notation `p.age >= 0`

Navigation expressions in typed FOL look exactly as in JAVA/JML

# Modeling Attributes in FOL

## Properties of attributes

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

## Changing the value of attributes

How to translate assignment to attribute `p.age=17;`?

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

Admit on left-hand side of update program location expressions

# Modeling Attributes in FOL

## Properties of attributes

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

## Changing the value of attributes

How to translate assignment to attribute `p.age=17;`?

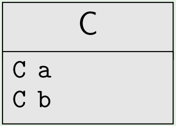
$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{p.age} := 17\}\langle\mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\mathtt{p.age\ =\ 17;\ rest}\rangle\phi, \Delta}$$

Admit on left-hand side of update program location expressions

# A Warning

Computing the effect of updates with attribute locations is complex

## Example

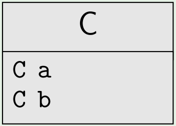



- Signature $FSym_{nr}$: `C a(C);` `C b(C);` `C o;`

# A Warning

**Computing the effect of updates with attribute locations is complex**

## Example

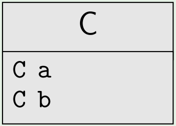



- Signature $\mathrm{FSym}_{nr}$: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a} := \mathtt{o}\}\{\mathtt{o.b} := \mathtt{o.a}\}$

# A Warning

Computing the effect of updates with attribute locations is complex

## Example

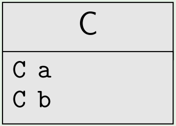



- Signature $\mathrm{FSym}_{nr}$: `C a(C);  C b(C);  C o;`
- Consider $\{\texttt{o.a} := \texttt{o}\}\{\texttt{o.b} := \texttt{o.a}\}$
- First update may affect left side of second update

# A Warning

Computing the effect of updates with attribute locations is complex

## Example

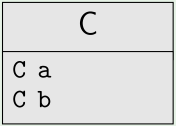



- Signature $\mathit{FSym}_{nr}$: `C a(C);  C b(C);  C o;`
- Consider $\{\texttt{o.a} := \texttt{o}\}\{\texttt{o.b} := \texttt{o.a}\}$
- First update may affect left side of second update
- `o.a` and `o.b` might refer to same object (be aliases)

# A Warning

Computing the effect of updates with attribute locations is complex

## Example




- Signature $\mathrm{FSym}_{nr}$: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a} := \mathtt{o}\}\{\mathtt{o.b} := \mathtt{o.a}\}$
- First update may affect left side 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

# Modeling Static Attributes in FOL

## Modeling class (static) attributes

For each class $C$ with static attribute a of type $T$:
$\mathrm{FSym}_{nr}$ declares flexible constant $T$ a;

- Value of a is $\mathcal{I}(\mathtt{a})$ for all instances of $C$
- If necessary, qualify with class (path):
  **byte** `java.lang.Byte.MAX_VALUE`
- Standard values are predefined in KeY:
  $\mathcal{I}(\mathtt{java.lang.Byte.MAX\_VALUE}) = 127$

# The Self Reference

## Modeling reference **this** to the receiving object

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

in JML: `Object self;`

in JAVA: `Object` **this**`;`

in KeY: `Object self;`

Default assumption in JML-KeY translation: $!(\texttt{self} = \textbf{null})$

# Which Objects do Exist?

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

# 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

$$\models \forall\ T\ x;\ \phi \text{ -> } [\mathrm{p}](\forall\ T\ x;\ \phi) \qquad \text{is valid for rigid } \phi$$

# 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

$$\models \forall\, T\ x;\ \phi \texttt{ -> } [\mathrm{p}](\forall\, T\ x;\ \phi) \qquad \text{is valid for rigid } \phi$$

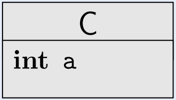
## Realizing Constant Domain Assumption

- flexible function **boolean** `<created>(Object);`
- Equal to **true** iff argument object has been created
- Initialized as $\mathcal{I}(\texttt{<created>})(o) = F$ for all $o \in \mathcal{D}$
- Object creation modeled as $\{\texttt{o.<created>} := \textbf{true}\}$ for next "free" o

# Outline

# Quantified Updates

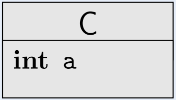
## Initialization of all objects in a given class C




- Specify that default value of attribute **int** a(C) is 0

# Quantified Updates

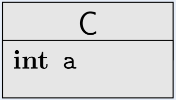
## Initialization of all objects in a given class `C`




- Specify that default value of attribute **int** `a(C)` is 0
- Can use $\forall$ `C o; o.a` $\doteq$ `0` in premise

# Quantified Updates

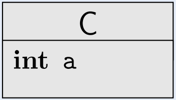
## Initialization of all objects in a given class `C`




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

# 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 $\doteq 0$ in premise
- Problem: difficult to exploit for update simplification

## Definition (Quantified Update)

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

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

- For all objects $d$ in $\mathcal{D}^{\mathrm{T}}$ such that $\beta_{\mathtt{x}}^{d} \models \mathtt{P}$
  perform the updates $\{\mathtt{l} := \mathtt{r}\}$ under $\beta_{\mathtt{x}}^{d}$ in parallel
- If there are several l with conflicting $d$ then choose $T$-minimal one

# Quantified Updates

- 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

# Quantified Updates

- 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)

```
\exists int n; ({\for int i; l := i}(l = n))
```

- Is valid both for JAVA **int** and $\mathbb{Z}$ ($n \doteq 0$ non-standard order)
- Proven automatically by update simplifier
- `lect13/update.key` Demo

# Quantified Updates

- 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)**

```
\exists int n; ({\for int i; l := i}(l = n))
```

- Is valid both for JAVA **int** and $\mathbb{Z}$ ($n \doteq 0$ non-standard order)
- Proven automatically by update simplifier
- `lect13/update.key` Demo

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

```
{\for T o; o.a := 0}
```

# Outline

# Extending Dynamic Logic to Java

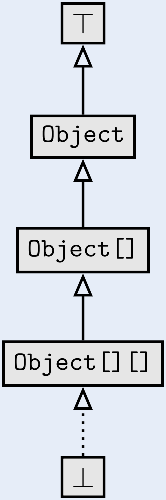
## Any syntactically correct JAVA program (plus some extensions)

- 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 (meta-programming)
- API method calls: need either JML contract or implementation

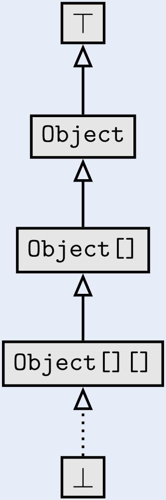
## Arrays




- JAVA type hierarchy includes array types that occur in given program

# JAVA Dynamic Logic: Arrays

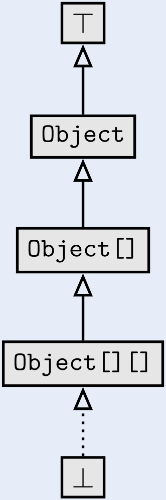
## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules

# JAVA Dynamic Logic: Arrays

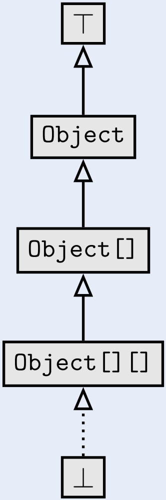
## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- flexible functions modeling attributes can have array type

# JAVA Dynamic Logic: Arrays

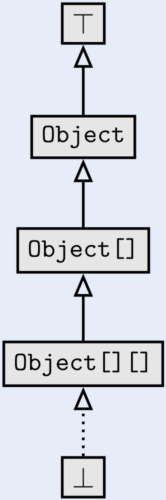
## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- flexible functions modeling attributes can have array type
- Value of entry in array `T[] ar;` defined in class `C` depends on reference `ar` to array in `C` and index

# JAVA Dynamic Logic: Arrays

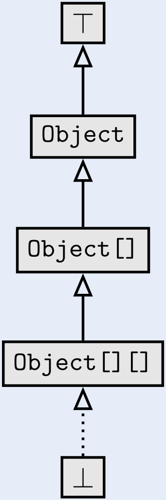
## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- flexible functions modeling attributes can have array type
- Value of entry in array `T[] ar;` defined in class `C` depends on reference `ar` to array in `C` and index
- Model array with flexible function `T []`(`C`,**int**)

# Java Dynamic Logic: Arrays

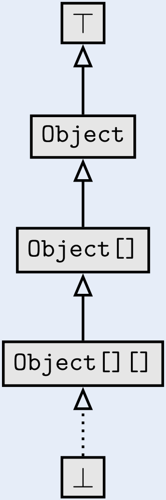
## Arrays




- Java type hierarchy includes array types that occur in given program
- Types ordered according to Java subtyping rules
- flexible functions modeling attributes can have array type
- Value of entry in array `T[] ar;` defined in class `C` depends on reference `ar` to array in `C` and index
- Model array with flexible function `T []`(`C`,**int**)
- Instead of `[](ar,i)` write `ar[i]`

# JAVA Dynamic Logic: Arrays

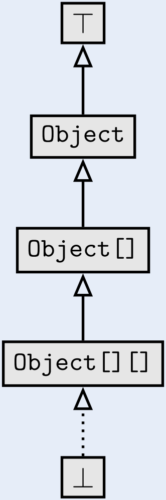
## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- flexible functions modeling attributes can have array type
- Value of entry in array `T[] ar;` defined in class `C` depends on reference `ar` to array in `C` and index
- Model array with flexible function `T []( C ,` **int** `)`
- Instead of `[](ar,i)` write `ar[i]`
- Arrays `a` and `b` can refer to same object (aliases)

# JAVA Dynamic Logic: Arrays

## Arrays




- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- flexible functions modeling attributes can have array type
- Value of entry in array `T[] ar;` defined in class `C` depends on reference `ar` to array in `C` and index
- Model array with flexible function `T []`(`C`,**int**)
- Instead of `[](ar,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 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)

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

# Complex Expressions by Symbolic Execution

## Decomposition of complex terms by symbolic execution

Follow the rules laid down in JAVA Language Specification

### Local code transformations

$$\text{evalOrderIteratedAssgnmt}\quad \frac{\Gamma \Longrightarrow \langle \texttt{y = t; x = y; rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \texttt{x = y = t; rest}\rangle\phi, \Delta}\quad \texttt{t}\ \text{simple}$$

### Temporary variables store result of evaluating subexpression

$$\text{ifEval}\quad \frac{\Gamma \Longrightarrow \langle \textbf{boolean}\ \texttt{v0; v0 = b;}\ \textbf{if}\ \texttt{(v0) p; r}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \textbf{if}\ \texttt{(b) p; r}\rangle\phi, \Delta}\quad \text{b complex}$$

Guards of conditionals/loops always evaluated (hence: side effect-free) before conditional/unwind rules applied

# Java Dynamic Logic: Abrupt Termination

## Abrupt Termination: Exceptions and Jumps

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

$$\langle \pi \ \textbf{try}\ \xi \ \ \texttt{p}\ \textbf{catch}\texttt{(e)}\ \texttt{q}\ \textbf{finally}\ \texttt{r;}\ \omega \rangle \phi$$

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

# Java Dynamic Logic: Abrupt Termination

## Abrupt Termination: Exceptions and Jumps

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

$$\langle \pi\ \mathbf{try}\ \xi\ \ \mathtt{p}\ \mathbf{catch}(\mathtt{e})\ \mathtt{q}\ \mathbf{finally}\ \mathtt{r;}\ \omega\rangle\phi$$

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

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

$$\frac{\Longrightarrow \langle \pi\ \mathbf{if}\ (\mathtt{e}\ \mathbf{instanceof}\ \mathtt{T})\ \{\mathbf{try}\ \mathtt{x=e;q}\ \mathbf{finally}\ \mathtt{r}\}\ \mathbf{else}\ \{\mathtt{r;}\ \mathbf{throw}\ \mathtt{e}\}\mathtt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \mathbf{try}\ \{\mathbf{throw}\ \mathtt{e;}\ \mathtt{p}\}\ \mathbf{catch}(\mathtt{T\ x})\ \mathtt{q}\ \mathbf{finally}\ \mathtt{r;}\ \omega\rangle\phi}$$

# JAVA Dynamic Logic: Abrupt Termination

## Abrupt Termination: Exceptions and Jumps

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

$$\langle \pi \ \textbf{try} \ \xi \ \ \texttt{p} \ \textbf{catch}(\texttt{e}) \ \texttt{q} \ \textbf{finally} \ \texttt{r;} \ \omega \rangle \phi$$

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

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

$$\frac{\Longrightarrow \langle \pi \ \textbf{if} \ (\texttt{e} \ \textbf{instanceof} \ \texttt{T}) \ \{\textbf{try} \ \texttt{x=e;q} \ \textbf{finally} \ \texttt{r}\} \ \textbf{else} \ \{\texttt{r;} \ \textbf{throw} \ \texttt{e}\}\texttt{;} \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ \textbf{try} \ \{\textbf{throw} \ \texttt{e;} \ \texttt{p}\} \ \textbf{catch}(\texttt{T x}) \ \texttt{q} \ \textbf{finally} \ \texttt{r;} \ \omega \rangle \phi}$$

**Demo**

`lect13/exc2.key`

# Java Dynamic Logic: Aliasing

## Reference Aliasing

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

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

# JAVA Dynamic Logic: Aliasing

## Reference Aliasing

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

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

## Unnecessary case analyses

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

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

# JAVA Dynamic Logic: Aliasing

## Reference Aliasing

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

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

## Unnecessary case analyses

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

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

## Updates avoid case analyses— Demo `lect13/alias2.key`

- Delayed state computation until clear what is required
- Eager simplification of updates

# Aliasing

## Form of JAVA program locations

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

## Assignment rule for arbitrary JAVA locations

$$\text{assign}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := t\}\langle \pi\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi \mathtt{l\ =\ t;}\ \omega\rangle\phi, \Delta}$$

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

$$\{\mathtt{o.a} := t\}\mathtt{u.a} \rightsquigarrow \backslash\mathit{if}\ (\{\mathtt{o.a} := t\}\mathtt{u} \doteq \mathtt{o})\ \backslash\mathit{then}\ (t)\ \backslash\mathit{else}\ (\{\mathtt{o.a} := t\}\mathtt{u}).\mathtt{a}$$

# JAVA Dynamic Logic: Method Calls

## Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \cdots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

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

## Actions of rule methodCall

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

# Method Calls

## Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_\texttt{i}$ `p#i =`*$arg_i$*;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=c)\{ body \}}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#0,...,p\#n)@C;}\ \omega \rangle \phi, \Delta}$$

# Method Calls

## Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_\texttt{i}\ \texttt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \texttt{method-frame(source=C, this=c)\{ body \}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \texttt{c.m(p\#0,...,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

Symbolic Execution

Only static information available, proof splitting if necessary

# Method Calls

## Method Body Expand

1. Execute code that binds actual to formal parameters $\mathtt{T_i}\ \mathtt{p\#i} = arg_i$;
2. Call rule methodBodyExpand

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

Symbolic Execution

Runtime infrastructure required in calculus

# Method Calls

## Method Body Expand

1. Execute code that binds actual to formal parameters $\mathtt{T_i}$ $\mathtt{p\#i} = arg_i$;
2. Call rule methodBodyExpand

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

Symbolic Execution
Runtime infrastructure required in calculus

## Demo

```
lect13/method2.key
```

# A Tour of JAVA Features in DL

## 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

# A Round Tour of JAVA Features in DL

## Null pointer exceptions

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

Need to model possibility that $\texttt{o} \doteq \textbf{null}$ in o.a

- KeY creates PO for $!\,\texttt{o} \doteq \textbf{null}$ upon each field access
- Can be switched off with option nullPointerPolicy

# A Round Tour of JAVA Features in DL

## Object initialization

JAVA has complex rules for object initialization

- Chain of constructor calls until Object
- Implicit calls to **super**()
- Visbility issues
- Initialization sequence

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

# A Round Tour of JAVA Features in DL

## 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

# A Round Tour of JAVA Features in DL

## 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`

# Outline

# 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

# Outline

# 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