Formal Specification and Verification

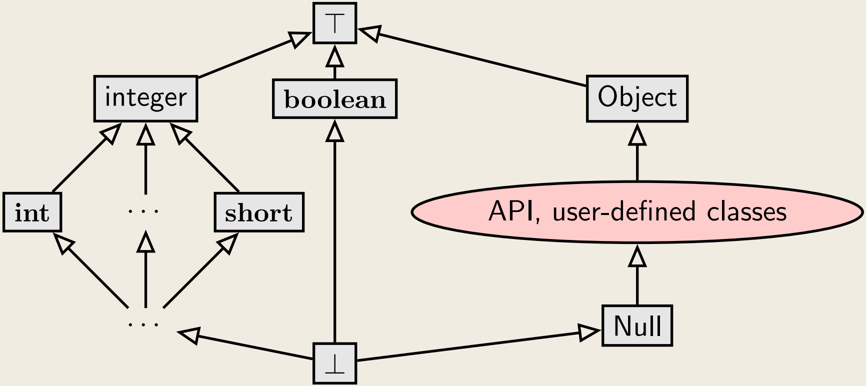
Reasoning about Java Programs

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at
Chalmers University, Göteborg

Java Type Hierarchy

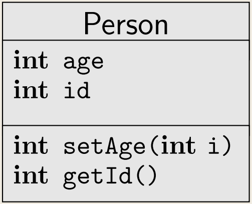
Signature based on Java's type hierarchy



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
int id

int setAge(int i)
int getId()
```

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

Modelling Attributes in FOL

Modeling instance attributes

Person

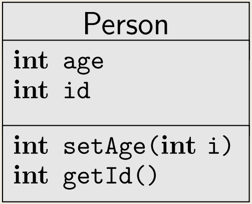
```
int age
int id

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**

Modelling Attributes in FOL

Modeling instance attributes



Person

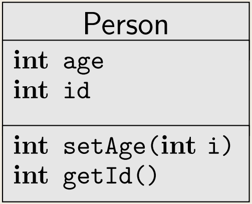
`int age`
`int id`

`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

Modelling Attributes in FOL

Modeling instance attributes



Person

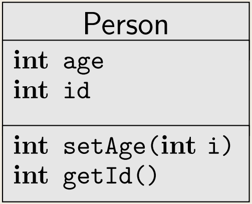
```
int age
int id

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: FSym_{nr} declares non-rigid function $T\ \mathsf{a}(C)$;

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
- $\mathcal{I}$(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);

Attribute Access

Signature FSym_{nr}: **int** age(Person); Person `p`;

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

Typed FOL age(`p`)>=0

KeY postfix notation `p.age >= 0`

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

Modeling Attributes in FOL Cont'd

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;`?

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

Admit on left-hand side of update program location expressions

Modeling Attributes in FOL Cont'd

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;`?

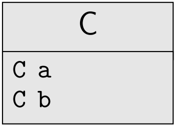
$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathtt{p.age} := 17\}\langle\mathtt{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\mathtt{p.age\ =\ 17;\ rest}\rangle\phi, \Delta}$$

Admit on left-hand side of update program location expressions

A Warning

Computing the effect of updates with attribute locations is complex

Example



C

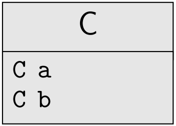
`C a`
`C b`

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

A Warning

Computing the effect of updates with attribute locations is complex

Example

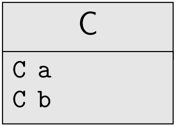


- Signature FSym_{nr}: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$

A Warning

Computing the effect of updates with attribute locations is complex

Example


C

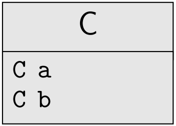
`C a`
`C b`

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

A Warning

Computing the effect of updates with attribute locations is complex

Example

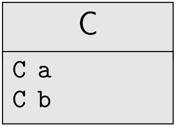


- Signature FSym_{nr}: `C a(C);   C b(C);   C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$
- First update may affect left side of second update
- `o.a` and `o.b` might refer to same object (be aliases)

A Warning

Computing the effect of updates with attribute locations is complex

Example



- Signature $FSym_{nr}$: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$
- 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:
FSym_{nr} declares non-rigid 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:
  $\mathcal{I}(\mathbf{byte}\ \mathtt{java.lang.Byte.MAX\_VALUE}) = 127$

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: $!(\texttt{self} = \mathbf{null})$

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

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

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

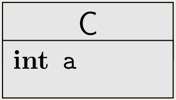
$\models \forall\ T\ x; \phi \ \texttt{->}\ [\mathrm{p}](\forall\ T\ x; \phi)$ is valid for rigid ϕ

Realizing Constant Domain Assumption

- Non-rigid 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

Quantified Updates

Initialization of all objects in a given class `C`



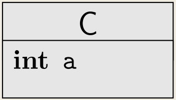
C

int a

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

Quantified Updates

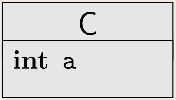
Initialization of all objects in a given class `C`



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

Quantified Updates

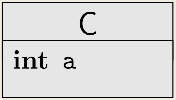
Initialization of all objects in a given class `C`



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

Definition (Quantified Update)

For `T` well-ordered type (no ∞ descending chains): quantified update:

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

- For all objects d in $\mathcal{D}^{\mathtt{T}}$ 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 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

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 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

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 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

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

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

Extending Dynamic Logic to Java

Any syntactically correct Java with some extensions

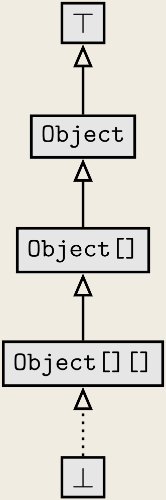
- Needs not be compilable unit
- Permit externally declared, non-initialized variables
- Referenced class definitions loaded in background

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

Java Features in Dynamic Logic: Arrays

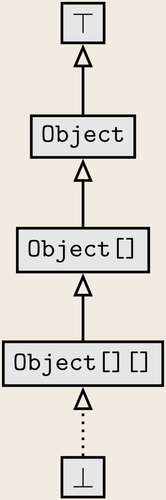
Arrays



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

Java Features in Dynamic Logic: Arrays

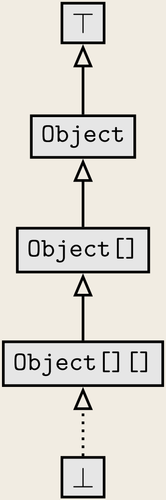
Arrays



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

Java Features in Dynamic Logic: Arrays

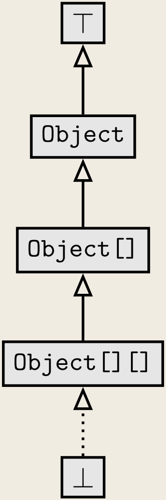
Arrays



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

Java Features in Dynamic Logic: Arrays

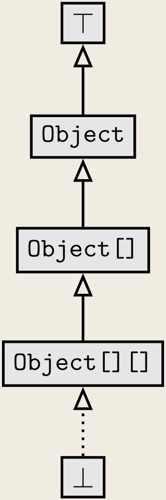
Arrays



- Java type hierarchy includes array types that occur in given program
- Types ordered according to Java subtyping rules
- Non-rigid 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 Features in Dynamic Logic: Arrays

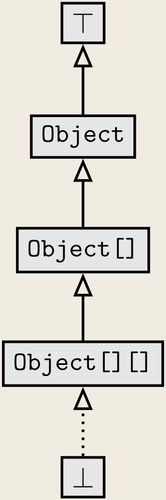
Arrays



- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- Non-rigid 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 non-rigid function `T []`(`C`,**int**)

Java Features in Dynamic Logic: Arrays

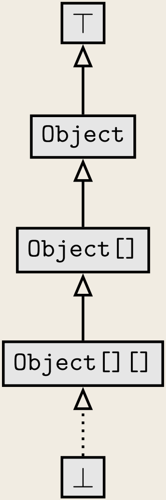
Arrays



- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- Non-rigid 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 non-rigid function `T []`(`C`,**int**)
- Instead of `[](ar,i)` write `ar[i]`

Java Features in Dynamic Logic: Arrays

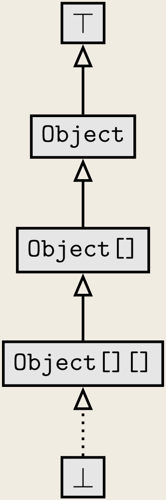
Arrays



- JAVA type hierarchy includes array types that occur in given program
- Types ordered according to JAVA subtyping rules
- Non-rigid 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 non-rigid function `T []`$($`C`$,$**int**$)$
- Instead of `[](ar,i)` write `ar[i]`
- Arrays `a` and `b` can refer to same object (aliases)

Java Features in Dynamic Logic: Arrays

Arrays



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

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

Complex Expressions Cont'd

Decomposition of complex terms by symbolic execution

Follow the rules laid down in JAVA Language Specification

Local code transformations

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

Temporary variables store result of evaluating subexpression

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

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

Java Features in 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

Java Features in 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 Features in 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

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

Demo

`lect13/exc2.key`

Java Features in Dynamic Logic: Aliasing

Reference Aliasing

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

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

Java Features in 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

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

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

Java Features in Dynamic Logic: Aliasing

Reference Aliasing

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

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

Unnecessary case analyses

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

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

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

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

Aliasing Cont'd

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 Features in Dynamic Logic: Method Calls

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

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

where `m` declared as **void** $\mathtt{m}(\mathtt{T}_0\ \mathtt{p}_0, \ldots, \mathtt{T}_\mathtt{n}\ \mathtt{p}_\mathtt{n})$

Actions of rule methodCall

- (type conformance of arg_i to $\mathtt{T}_\mathtt{i}$ guaranteed by JAVA compiler)
- for each formal parameter $\mathtt{p}_\mathtt{i}$ of `m`:
  declare & initialize new local variable $\mathtt{T}_\mathtt{i}\ \mathtt{p\#i} = arg_i\mathtt{;}$
- look up implementation class C of `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 Cont'd

Method Body Expand

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

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

Method Calls Cont'd

Method Body Expand

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

Only static information available, proof splitting

Method Calls Cont'd

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

Runtime infrastructure required in calculus

Method Calls Cont'd

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 Round Tour of Java Features in DL Cont'd

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 Cont'd

Null pointer exceptions

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

Need to model possibility that $\mathtt{o} \doteq \mathbf{null}$ in $\mathtt{o.a}$

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

A Round Tour of Java Features in DL Cont'd

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 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:
   - **2.1** Show that requires clause is satisfied
   - **2.2** Obtain postcondition from ensures clause
   - **2.3** Delete updates with modifiable locations from symbolic state

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:
   - 2.1 Show that requires clause is satisfied
   - 2.2 Obtain postcondition from ensures clause
   - 2.3 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`

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