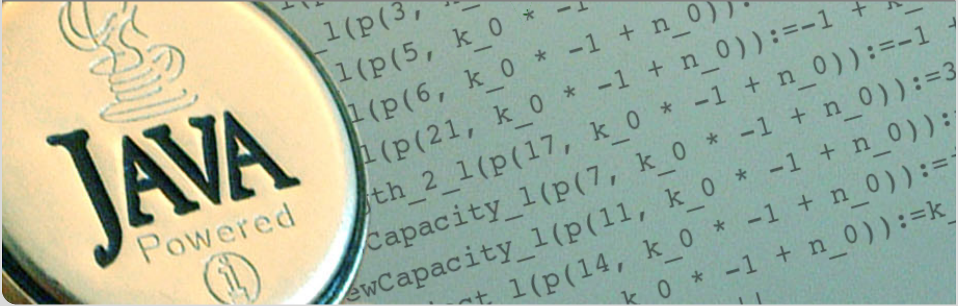
Specification & Formal Analysis of Java Programs

Functional Verification of Java Programs

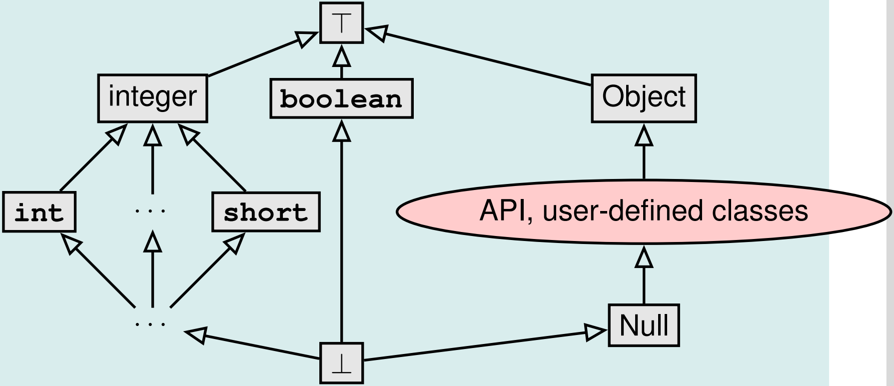
Prof. Dr. Bernhard Beckert | ADAPT 2010

KIT – Institut für Theoretische Informatik



Java Type Hierarchy

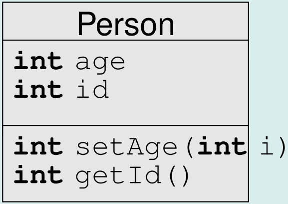
Signature based on Java's type hierarchy



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 $o \in \mathtt{D}^{\text{Person}}$ 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

Modelling Attributes in FOL

Modeling instance attributes

```
Person
int age
int id

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

- Each $o \in \mathtt{D}^{\mathrm{Person}}$ 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

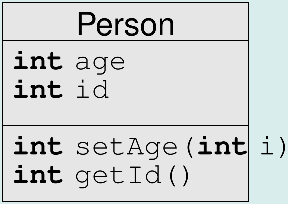
Signature FSym$_{nr}$: **`int`** `age`(Person); Person `p`;

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

Typed FOL `age`(`p`)>=0

Modelling Attributes in FOL

Modeling instance attributes



Person

int `age`
int `id`

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

- Each $o \in \mathtt{D}^{\text{Person}}$ has associated `age` value
- $\mathcal{I}(\mathtt{age})$ 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
  $T\ \mathtt{a}(C)$;

Attribute Access

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

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

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

Modelling Attributes in FOL

Modeling instance attributes

```
Person
int age
int id

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

- Each $o \in \mathrm{D}^{\text{Person}}$ 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

Modelling Attributes in FOL

Modeling instance attributes

Person

int age
int id

int setAge(**int** i)
int getId()

- Each $o \in \mathtt{D}^{\text{Person}}$ has associated `age` value
- $\mathcal{I}(\mathtt{age})$ 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
  $T\ \mathtt{a}(C)$;

Attribute Access

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

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

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

Modeling Attributes in FOL Cont'd

Properties of attributes

- When not initialized, $\mathcal{I}(\mathtt{a}) = \mathtt{\textbf{null}}$
- 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, $\mathcal{I}(\mathtt{a}) = \mathbf{null}$
- 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;` ?

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

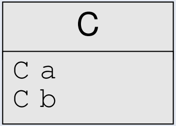
Admit on left-hand side of update *program location expressions*

A Warning

Computing the effect of updates with attribute locations is complex

Example

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

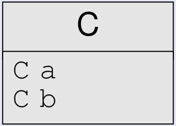


KeY applies rules automatically, you don't need to worry about details

A Warning

Computing the effect of updates with attribute locations is complex

Example



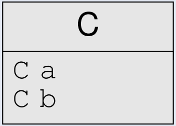
- Signature $FSym_{nr}$: `C a(C); C b(C); C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$
- First update may affect of second update
- `o.a` and `o.b` might refer to same object (be *aliases*)

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



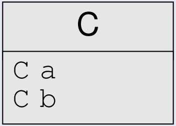
- Signature FSym_{nr}: `C a(C);` `C b(C);` `C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$
- First update may affect of second update
- `o.a` and `o.b` might refer to same object (be *aliases*)

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



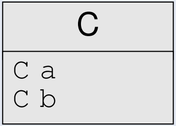
- Signature FSym$_{nr}$: `C a(C); C b(C); C o;`
- Consider $\{\mathtt{o.a.a} := \mathtt{o}\}\{\mathtt{o.b.a} := \mathtt{o.a}\}$
- First update may affect of second update
- `o.a` and `o.b` might refer to same object (be *aliases*)

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}: `C a(C); C b(C); C o;`
- Consider $\{\texttt{o.a.a} := \texttt{o}\}\{\texttt{o.b.a} := \texttt{o.a}\}$
- 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

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:
  $\mathcal{I}(\mathtt{\textbf{byte}\ java.lang.Byte.MAX\_VALUE}) = 127$

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`** ?

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

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

Realizing Constant Domain Assumption

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

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

Realizing Constant Domain Assumption

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

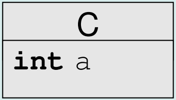
$$\models \forall\, T\ x;\ \phi \ \texttt{->}\ [\mathrm{p}](\forall\, T\ x;\ \phi) \qquad \text{is valid for rigid } \phi$$

Realizing Constant Domain Assumption

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

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
- Can use $\forall\, \mathtt{C\ o;\ o.a} \doteq 0$ in premise
- *Problem:* difficult to exploit for update simplification

Definition (Quantified Update)

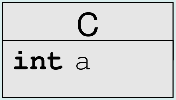
For $\mathtt{T}$ well-ordered type (no ∞ descending chains): *quantified update*:

`{\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

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
- Can use $\forall\, \mathtt{C\ o;\ o.a} \doteq 0$ in premise
- *Problem:* difficult to exploit for update simplification

Definition (Quantified Update)

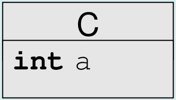
For `T` well-ordered type (no ∞ descending chains): *quantified update*:

$$\{\backslash\mathbf{for}\ \mathtt{T\ x;}\ \backslash\mathbf{if}\ \mathtt{P;\ l := r}\}$$

- *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

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
- Can use $\forall$ `C o; o.a` $\doteq$ 0 in premise
- *Problem:* difficult to exploit for update simplification

Definition (Quantified Update)

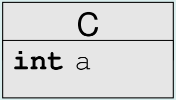
For `T` well-ordered type (no ∞ descending chains): *quantified update*:

`{`**`\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

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
- Can use $\forall\,\mathtt{C\ o;\ o.a} \doteq 0$ 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

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}
```

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}
```

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

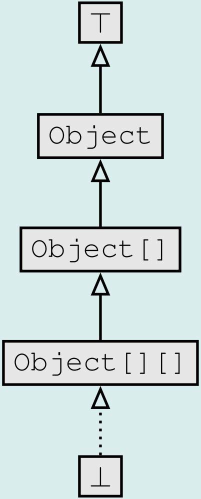
- Needs not be compilable unit
- Permit externally declared, non-initialized variables
- Referenced class definitions loaded in background

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

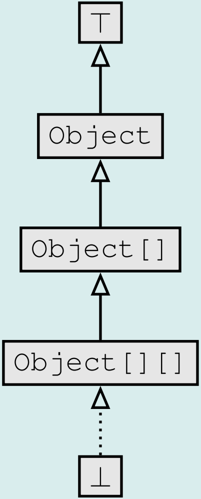
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

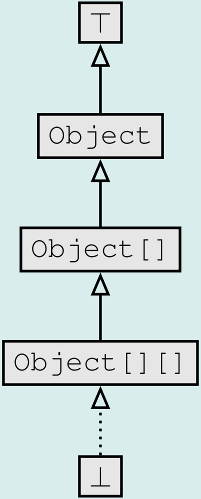
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

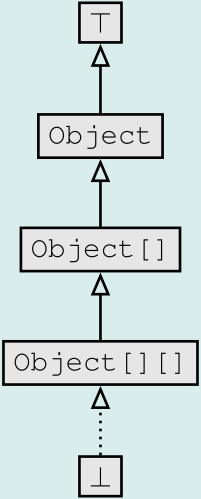
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

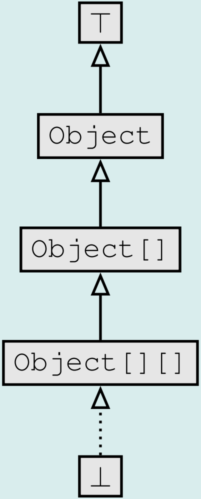
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

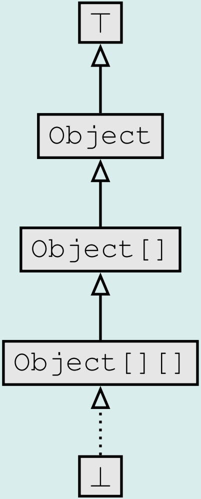
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

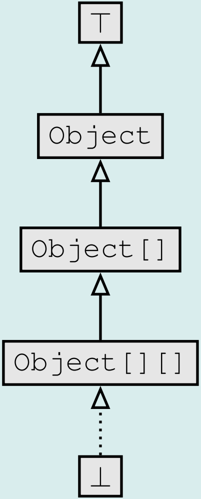
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

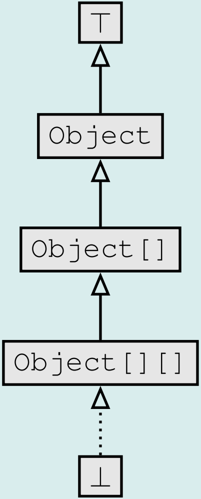
Arrays



- Java type hierarchy includes array types

Java Features in Dynamic Logic: Arrays

Arrays



- Java type hierarchy includes array types

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

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

Temporary variables store result of evaluating subexpression

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

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*

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

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

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

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

Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

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

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

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

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

Java Features in Dynamic Logic: Abrupt Termination

Abrupt Termination: Exceptions and Jumps

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

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

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

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

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

Java Features in Dynamic Logic: Aliasing

Reference Aliasing

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

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

Unnecessary case analyses

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

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

Updates avoid case analyses— Demo

`lect13/alias2.key`

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

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

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

Updates avoid case analyses— Demo

`lect13/alias2.key`

Java Features in Dynamic Logic: Aliasing

Reference Aliasing

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

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

Unnecessary case analyses

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

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

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

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

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

$\{o.a := t\}_{\mathcal{U}}\ a \rightsquigarrow \backslash\mathit{if}\,(\{o.a := t\}_{\mathcal{U}} \doteq$

Java Features in Dynamic Logic: Method Calls

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

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

where `m` declared as **`void`** `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 `m`:
  declare & initialize new local variable $\mathtt{T}_i\ \mathtt{p\#i} = arg_i;$
- look up *implementation* class C of `m` and split proof if implementation cannot be uniquely determined
- create *method invocation* $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})@C$

Method Calls Cont'd

Method Body Expand

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

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

Demo

`lect13/method2.key`

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters
   `Ti p#i =`*arg$_i$*;
2. Call rule *methodBodyExpand*

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

Symbolic Execution
Only static information available, proof splitting

Demo

`lect13/method2.key`

Method Calls Cont’d

Method Body Expand

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

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

Symbolic Execution
Runtime infrastructure required in calculus

Demo

`lect13/method2.key`

Method Calls Cont'd

Method Body Expand

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

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

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 $\mathrm{o} \doteq \texttt{null}$ in $\mathrm{o.a}$

- KeY creates PO for $!\,\mathrm{o} \doteq \texttt{null}$ 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:
   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.

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