Selected Topics

Dependency Contracts

Accessible Clause in JML

Accessible Clause (introduced in previous lecture)

Defines *dynamic frame* for invariants, model fields and methods

//@ **accessible \inv**: <*JML expression of type* **\locset**>

//@ **accessible <model field>**: <*JML expression of type* **\locset**>

(or attached to a method)

How can we

- verify that a specified accessible clause is respected by the program (i.o.w. that it is correct)?
- use an accessible clause to ease verification of a program?

Correctness of an Accessible Clause

TECHNISCHE UNIVERSITÄT DARMSTADT

Intuitive Meaning

The value of an invariant, model field or method return value should only depend on the locations specified by the accessible clause.

Reminder: **\invariant_for**(o) (with o JML expression of type T) translated to

$$T\text{::inv}(\text{heap}, o)$$

(+ rules that replace $T\text{::inv}(\text{heap}, o)$ by the conjunction of all instance invariants of type T)

Software Engineering Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Correctness of an accessible clause acc for an **invariant**

Let acc(heap, self) = E(<**accessible** clause>) be a DL term of type LocSet

$$\Big(\mathrm{wellFormed(h)} \land \mathrm{wellFormed(heap)} \land T{::}\mathbf{inv}(\mathrm{heap}, \mathrm{self}) \land$$
$$(\neg \mathrm{self} \doteq \mathrm{null} \land \mathrm{boolean{::}select(self,\langle created\rangle)} \doteq \mathrm{TRUE}) \land$$
$$\forall\, \mathrm{Object}\ o;\ \forall\, \mathrm{Field}\ f;\ ($$
$$\mathrm{singleton}(o,f) \subseteq \mathrm{acc(heap, self)}$$
$$\lor\ \mathrm{boolean{::}select}(\mathrm{heap}, o, \langle\mathrm{created}\rangle) \doteq \mathrm{FALSE}$$
$$\lor\ \mathrm{any{::}select}(\mathrm{heap}, o, f) \doteq \mathrm{any{::}select}(\mathrm{h}, o, f)\ )$$
$$\Big) \rightarrow \Big(\ T{::}\mathbf{inv}(\mathrm{heap}, \mathrm{self}) \leftrightarrow \{\mathrm{heap{:=}h}\}\ T{::}\mathbf{inv}(\mathrm{heap}, \mathrm{self})\ \Big)$$

(with self:T, h:Heap be program variables of type $T \leq$Object and Heap)

Software Engineering Group

Correctness of an Accessible Clause
Model Field

Correctness of an accessible clause acc for a model field $\mathbf{mf}$

Let $\mathrm{acc}(\mathrm{heap}, \mathrm{self}) = \mathrm{E}(<\mathbf{accessible}\ \mathrm{clause}>)$ be a DL term of type LocSet

$$
\begin{aligned}
&\big(\mathrm{wellFormed}(\mathrm{heap}) \land \mathrm{wellFormed}(\mathrm{h}) \land T{::}\mathbf{inv}(\mathrm{heap}, \mathrm{self}) \land \\
&\quad (\neg \mathrm{self} \doteq \mathrm{null} \land \mathrm{boolean{::}select}(\mathrm{self}, <\mathrm{created}>) \doteq \mathrm{TRUE}) \\
&\land \forall \mathrm{Object}\ o; \forall \mathrm{Field}\ f; (\\
&\quad \mathrm{singleton}(o,f) \subseteq \mathrm{acc}(\mathrm{heap}, \mathrm{self}) \\
&\quad \lor \mathrm{boolean{::}select}(\mathrm{heap}, o, <\mathrm{created}>) \doteq \mathrm{FALSE} \\
&\quad \lor \mathrm{any{::}select}(\mathrm{heap}, o, f) \doteq \mathrm{any{::}select}(\mathrm{h}, o, f)) \\
&\big) \rightarrow \big(T{::}\mathbf{mf}(\mathrm{heap}, \mathrm{self}) \doteq (\{\mathrm{heap}:=\mathrm{h}\}\ T{::}\mathbf{mf}(\mathrm{heap}, \mathrm{self})\ )\big)
\end{aligned}
$$

(with self$:T$, $\mathrm{h}:\mathrm{Heap}$ be program variables of type $T \leq \mathrm{Object}$ and Heap)

Correctness of an Accessible Clause
Query Methods

TECHNISCHE UNIVERSITÄT DARMSTADT

Correctness of an accessible clause acc for a query method **m**

Let acc(heap, self, *args*) = E(<**accessible** clause>) be a DL term of type LocSet

$$
\begin{aligned}
&\big(\ \mathrm{pre} \wedge \mathrm{freePre} \wedge \mathrm{wellFormed}(h) \wedge \\
&\quad (\forall \mathrm{Object}\ o; \forall \mathrm{Field}\ f; (\\
&\qquad\quad \mathrm{any::select}(\mathrm{heap}, o, f) \doteq \mathrm{any::select}(h, o, f) \\
&\qquad \vee\ \mathrm{singleton}(o, f) \subseteq \mathrm{acc}(\mathrm{heap}, \mathrm{self}, args) \\
&\qquad \vee\ \mathrm{boolean::select}(\mathrm{heap}, o, \langle\mathrm{created}\rangle) \doteq \mathrm{FALSE}) \\
&\quad \wedge [\mathrm{res} = \mathrm{self}.\mathbf{m}(p_1, \ldots, p_n)@C;]\ \mathrm{res} \doteq \mathrm{r}_1 \\
&\quad \wedge \{\mathrm{heap} := h\}[\mathrm{res} = \mathrm{self}.\mathbf{m}(p_1, \ldots, p_n)@C;]\ \mathrm{res} \doteq \mathrm{r}_2\ \big) \\
&\rightarrow \mathrm{r}_1 \doteq \mathrm{r}_2
\end{aligned}
$$

Software Engineering Group

Dependency Contracts
Motivation

TECHNISCHE UNIVERSITÄT DARMSTADT

How does it help the verification?

$$\mathrm{inv}(h_1, u) \;\; ==> \;\; \mathrm{inv}(h_2, u)$$

where inv is framed by location set $\{(o, attr)\}$

If we can prove that

- h and h' only differ in locations ***other*** than $(o, attr)$

then

- $\mathrm{inv}(h_2, u)$ can be replaced by $\mathrm{inv}(h_1, u)$

and the sequent can be closed directly

(otherwise, inv needs to be expanded)

Dependency Contract
Invariants only

UseDependencyContract

$$\frac{\Gamma\ ,\text{guard} \rightarrow (T\text{::inv}(h_1,u) \leftrightarrow T\text{::inv}(h_2,u)) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

where

- $\text{acc}(h_1, u)$ is the translated accessible clause for $T\text{::inv}$
- guard is defined as

Formula that expresses that all objects created in heap h_1 also are created in heap h_2

$$\text{wellFormed}(h_1) \wedge \text{wellFormed}(h_2) \wedge \text{noDeallocation}(h_1, h_2)$$
$$\wedge\ T\text{::inv}(\text{heap}, \text{self}) \wedge (\neg(u \doteq \text{null}) \wedge \text{boolean::select}(u,\text{<created>}) \doteq \text{TRUE})$$
$$\wedge\ \forall \text{Object } o; \forall \text{Field } f; (\text{singleton}(o,f) \subseteq \text{acc}(h1, u)$$
$$\vee\ \text{select}(h1, o, \text{<created>}) \doteq \text{FALSE} \vee \text{select}(h1,o,f) \doteq \text{select}(h2,o,f)))$$

Demo

see:

Client.java (method m())

Integer Semantics

TO OVERFLOW OR NOT TO OVERFLOW

Integer Semantics

$$i > 0 \rightarrow \langle\, i = i + 1; \,\rangle\, i > 0$$

Not true in Java, if i=Integer.MAX_VALUE

Can we prove it?

Yes, because up-to-now Java integer operations +,-, /, % etc. were treated as their mathematical counterparts on $\mathbb{Z}$

Demo: BinarySort#magic(int)

Integer Semantics
Expressing Java int-Operators

Java Operator	Mathematical Interpretation
left + right	add(left, right)
left * right	mul(left, right)
left / right	jdiv(left, right)

similar for $\%$ which is in Java the remainder (jmod) not the mathematical modulo (mod, $\%$)

- add, mul pretty printed infix as '+', ' * ' (in their standard mathematical meaning)
- **Attention:** jdiv ≠ div (the latter one is pretty printed as /) both are division on $\mathbb{Z}$ (i.e., **no** overflow), but
  - div: euclidean division (rounds to next lower number)
    - div(4,2) = 2, div(4,3) = 1, div(5,2) = 2, div(-4,3) = -2, div(-5,2) = -3
  - jdiv: rounds towards zero (rounding as in Java)
    - jdiv(4,2) = 2, jdiv(4,3) = 1, jdiv(5,2) = 2, jdiv(-4,3) = -1, jdiv(-5,2) = -2

Integer Semantics
Expressing Java int-Operators

Java Operator	Mathematical Interpretation	Java Interpretation
left + right	add(left, right)	addJint(left, right) addJlong(left,right)
left * right	mul(left, right)	mulJint(left, right) mulJlong(left,right)
left / right	jdiv(left, right)	divJint(left, right) divJlong(left,right)

- addJint/mulJint/divJint(left, right) are defined as moduloInt(add/mul/jdiv(left, right))
- addJlong/mulJlong/divJlong(left, right) are defined as
  moduloLong(add/mul/jdiv(left, right))
- moduloInt(t) is defined as

  int_MIN + (int_HALFRANGE + t) % int_RANGE

  (with int_RANGE=4294967296 and int_HALFRANGE=2147483648)
- moduloLong(t) similar moduloInt using long_RANGE and long_HALFRANGE

Translation of Integer Operations
All Integer Semantics

$$\frac{i > 0 \implies \{i := \text{javaAddInt}(i, 1)\}\langle\rangle\, i > 0}{i > 0 \implies \langle\, i = i + 1;\, \rangle\, i > 0}$$

addition (rewrite rule)

$$\langle\ \text{var} = \text{left} + \text{right};\ \rangle\ \Phi \rightsquigarrow \{\text{var:=javaAddInt(left, right)}\}\ \langle\rangle\ \Phi$$

for maximum type of left, right being int (s.f. type promotion in Java)

In case of left or right being of type long: javaAddLong
(similar for other int operations)

Translation of Integer Operations

JavaDL$_{math}$

Function add is pretty printed infix and as '+'

$$\dfrac{\dfrac{i > 0 \implies \{\mathrm{i} := \mathrm{add}(\mathrm{i}, 1)\}\langle\,\rangle\, i > 0}{i > 0 \implies \{\mathrm{i} := \mathrm{javaAddInt}(\mathrm{i}, 1)\}\langle\,\rangle\, i > 0}}{i > 0 \implies \langle\, i = i + 1; \,\rangle\, i > 0}$$

For our current semantics (called JavaDL$_{math}$)

translateJavaAddInt (rewrite rule)

$$\mathrm{javaAddInt}(\mathrm{left},\mathrm{right}) \rightsquigarrow \mathrm{add}(\mathrm{left}, \mathrm{right})$$

translateJavaAddLong (rewrite rule)

$$\mathrm{javaAddLong}(\mathrm{left},\mathrm{right}) \rightsquigarrow \mathrm{add}(\mathrm{left}, \mathrm{right})$$

Translation of Integer Operations
$\text{JavaDL}_{\text{Java}}$

$$\frac{\dfrac{i > 0 \implies \{i := \text{addJint}(i, 1)\}\langle\,\rangle\, i > 0}{i > 0 \implies \{i := \text{javaAddInt}(i, 1)\}\langle\,\rangle\, i > 0}}{i > 0 \implies \langle\, i = i + 1;\, \rangle\, i > 0}$$

For Java semantics (called $\text{JavaDL}_{\text{Java}}$)

translateJavaAddInt (rewrite rule)

$$\text{javaAddInt}(\text{left}, \text{right}) \rightsquigarrow \text{addJint}(\text{left}, \text{right})$$

translateJavaAddLong (rewrite rule)

$$\text{javaAddLong}(\text{left}, \text{right}) \rightsquigarrow \text{addJlong}(\text{left}, \text{right})$$

Translation of Integer Operations
JavaDL$_{\text{CheckingOverflow}}$

$$\dfrac{\dfrac{i > 0 \implies \{i := \backslash\mathbf{if}\ (\mathrm{inInt}(\mathrm{add}(i, 1)))\ \backslash\mathbf{then}\ (\mathrm{add}(i, 1))\ \backslash\mathbf{else}\ (\mathrm{javaAddIntOverflow}(i,1))\}\langle\,\rangle\, i > 0}{i > 0 \implies \{i := \mathrm{javaAddInt}(i, 1)\}\langle\,\rangle\, i > 0}}{i > 0 \implies \langle\, i = i + 1;\,\rangle\, i > 0}$$

For checking overflow semantics (called JavaDL$_{\text{CheckingOverflow}}$)

translateJavaAddInt (rewrite rule)

unspecified ***function***, i.e., value may depend on left and right, but nothing more known

$$\mathrm{javaAddInt}(left,right) \rightsquigarrow$$

$$\backslash\mathbf{if}\ (\mathrm{inInt}(\mathrm{add}(left, right)))\ \backslash\mathbf{then}\ (\mathrm{add}(left, right))\ \backslash\mathbf{else}\ (\mathrm{javaAddIntOverflow}(left,right))$$

Translation of Integer Operations

$\text{JavaDL}_{\text{CheckingOverflow}}$

$i > 0 \implies$

$w(i,1))\}$

$\langle\ \rangle\, i > 0$

Intuitively $\text{JavaDL}_{\text{checking}}$ has the effect that only programs where no overflows can happen or where the actual value of an expression with overflow does not effect the property to be proven.

For checking overflow semantics (called $\text{JavaDL}_{\text{CheckingOverflow}}$)

translateJavaAddInt (rewrite rule)

unspecified ***function***, i.e., value may depend on left and right, but nothing more known

$\text{javaAddInt}(\text{left},\text{right}) \rightsquigarrow$

$\textbf{\textbackslash if}\ (\text{inInt}(\text{add}(\text{left}, \text{right})))\ \textbf{\textbackslash then}\ (\text{add}(\text{left}, \text{right}))\ \textbf{\textbackslash else}\ (\text{javaAddIntOverflow}(\text{left},\text{right}))$

Comparison Integer Semantics

TECHNISCHE UNIVERSITÄT DARMSTADT

Semantics	*Sound*	*Complete*	*Remarks*
$\text{JavaDL}_{\text{math}}$	no	no	**Usage**: teaching, prototyping proofs
$\text{JavaDL}_{\text{Java}}$	yes	yes	more complicated proofs, automation less powerful
$\text{JavaDL}_{\text{checking}}$	yes	no	detection of unintended over-/ underflow, usually as good automation as $\text{JavaDL}_{\text{math}}$, use if program should not have overflows

Software Engineering Group

Examples: Integer Semantics

$$i < 0 \Longrightarrow \langle i = i * 2; \rangle\, i < 0$$

Provable in

- $\mathrm{JavaDL}_{\mathrm{math}}$
  - yes
- $\mathrm{JavaDL}_{\mathrm{Java}}$
  - no, due to an underflow i might become positive
- $\mathrm{JavaDL}_{\mathrm{checking}}$
  - no, an underflow might happen

Examples: Integer Semantics

$$i > 0 \Longrightarrow \langle \texttt{i = i*2; i = (i < 0 ? 0 : i);} \rangle \ \texttt{i<=Integer.MAX\_VALUE}$$

Provable in

- $\text{JavaDL}_{\text{math}}$
  - no, not true for any initial value of i

    e.g., not true for $i > \texttt{Integer.MAX\_VALUE}/2$
- $\text{JavaDL}_{\text{Java}}$
  - yes, in case of an overflow i is set to 0 by conditional
- $\text{JavaDL}_{\text{checking}}$
  - no, an overflow might happen

Examples: Integer Semantics

$$i > 0 \Longrightarrow \langle i = i*2;\ \ i = (i < 0\ ?\ 0 : i); \rangle\ i*0 \doteq 0$$

Provable in

- $\mathrm{JavaDL}_{\mathrm{math}}$
  - yes
- $\mathrm{JavaDL}_{\mathrm{Java}}$
  - yes
- $\mathrm{JavaDL}_{\mathrm{checking}}$
  - yes (property to be shown is independent of value of i)

JML: \bigint

Reminder: Translation of JML to DL uses $\mathrm{javaAddInt}$ etc.

Hence, JML semantics depends on chosen integer semantics

JML type **\bigint** can be used to declare/cast expressions to unbounded integers e.g., JML expression

$$i + (\textbf{\textbackslash bigint})j$$

is of result type **\bigint** i.e. no overflows occurs here in any chosen semantics.

In fact, some JML expressions like reach are actually on **\bigint**, e.g.,

\reach: \locset × Object × Object × **\bigint**

Support in KeY

All three integer semantics supported

Selectable in Options | Taclet Options