This Part

making the connection between

JML

and

Dynamic Logic / KeY
This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,
- and proving

DL proof obligations from JML specifications
public class A {
    /*@ public normal_behavior */
    @ requires <Precondition>;
    @ ensures <Postcondition>;
    @ assignable <locations>;
    @*/
    public int m(params) {..}
}
From JML Contracts to Intermediate Format to Proof Obligations (PO)

```
public class A {
    /*@ public normal_behavior
        @ requires <Precondition>;
        @ ensures <Postcondition>;
        @ assignable <locations>;
        @*/
    public int m(params) {..}
}
```

Intermediate Format
(pre, post, div, var, mod)

Translation
From JML Contracts to Intermediate Format to Proof Obligations (PO)

```java
public class A {
    /*@ public normal_behavior
       @ requires <Precondition>;
       @ ensures <Postcondition>;
       @ assignable <locations>;
       @*/
    public int m(params) { .. }
}
```

Intermediate Format

\[ \text{Pre} \rightarrow \langle \text{this.m(params)}; \rangle \]

\( \text{(post & frame)} \)
## JML Translation: Normalizing JML Contracts

### Normalization of JML Contracts

1. Flattening of nested specifications
2. Making implicit specifications explicit
3. Processing of modifiers
4. Adding of default clauses if not present
5. Contraction of several clauses

The following introduces principles of this process.
Nested Specification Cases

nested specification cases allow to factor out common preconditions

@ public normal_behavior
@ requires R;
@

@ requires R1;
@ ensures E1;
@ assignable A1;
@

@ also
@

@ requires R2;
@ ensures E2;
@ assignable A2;
@ |}

expands to ... (next page)
Nested Specification Cases

(previous page) ... expands to

@ public normal_behavior
@ requires R;
@ requires R1;
@ ensures E1;
@ assignable A1;
@
@
@ also
@
@
@ public normal_behavior
@ requires R;
@ requires R2;
@ ensures E2;
@ assignable A2;
Nested Specification Cases

@ public normal_behavior
@ requires amount > 0;
@ 
@ requires amount + balance < limit && isValid() == true;
@ ensures \result == true;
@ ensures balance == amount + \old(balance);
@ assignable balance;
@ 
@ also
@ 
@ requires amount + balance >= limit;
@ ensures \result == false;
@ ensures unsuccessfulOperations == \old(unsuccessfulOperations) + 1;
@ assignable unsuccessfulOperations;
@ |}

expands to ... (next page)
Nested Specification Cases

(previous page) ... expands to

@ public normal_behavior
@ requires amount > 0;
@ requires amount + balance < limit && isValid() == true;
@ ensures \result == true;
@ ensures balance == amount + \old(balance);
@ assignable balance;
@
@
@ also
@

@ public normal_behavior
@ requires amount > 0;
@ requires amount + balance >= limit;
@ ensures \result == false;
@ ensures unsuccessfulOperations
@ == \old(unsuccessfulOperations) + 1;
@ assignable unsuccessfulOperations;
Normalisation:
Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- non_null by default
- Implicit \(\text{\textbackslash invariant\_for(this)}\) in requires, ensures & signals clause
Normalisation: Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- non_null by default
- Implicit `\invariant_for(this) in requires, ensures & signals` clause

Making 'kind of behavior' explicit

1. Deactivate implicit behavior specification:
   replace `normal_behavior/exceptional_behavior` by `behavior`
Normalisation: Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- non_null by default
- Implicit \ invariant_for(this) in requires, ensures & signals clause

Making 'kind of behavior' explicit

1. Deactivate implicit behavior specification:
   replace normal_behavior/exceptional_behavior by behavior
2. Add in case of replaced
   - normal_behavior the clause signals (Throwable t) false;
Normalisation:
Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- non_null by default
- Implicit \texttt{\textbackslash invariant\_for}(this) in requires, ensures & signals clause

Making 'kind of behavior' explicit

1. Deactivate implicit behavior specification:
   replace normal\_behavior/exceptional\_behavior by \texttt{behavior}

2. Add in case of replaced
   - normal\_behavior the clause \texttt{signals (Throwable t) false;}
   - exceptional\_behavior the clause \texttt{ensures false;}
Normalisation: Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- `non_null` by default
- Implicit `\text{invariant}_\text{for}(\text{this})` as requires, ensures & signals clause

Making `non_null` explicit for method specifications

1. Where `nullable` is absent, forbid null through preconditions (for parameters\(^a\)) and postcondition (for return value\(^a\)). E.g., for method `void m(\text{Object } o)` add `requires o \neq \text{null};`

2. Deactivate implicit `non_null` by adding `nullable`, where absent, to parameters\(^a\) and return type declarations\(^a\)

\(^a\)reference typed
Normalisation:
Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- non_null by default
- Implicit `\texttt{\textbackslash invariant\_for(this) as requires, ensures & signals clause}`

Making `\texttt{\textbackslash invariant\_for(this)}` explicit for method specifications

1. Add explicit `\texttt{\textbackslash invariant\_for(this) to non-helper method specs, as}
   - requires `\texttt{\textbackslash invariant\_for(this)}`;
   - ensures `\texttt{\textbackslash invariant\_for(this)}`;
   - signals (\texttt{Throwable t}) `\texttt{\textbackslash invariant\_for(this)}`;

2. Deactivate implicit `\texttt{\textbackslash invariant\_for(this) by adding helper modifier to method (if not already present)}`
Normalisation: Example

```java
/*@ public normal_behavior
   @ requires c.id >= 0;
   @ ensures \result == ( ... );
@*/
public boolean addCategory(Category c) {
```
Normalisation: Example

- Kind of behavior

```java
/*@
public behavior
@ requires c.id >= 0;
@ ensures \result == (...);
@ signals (Throwable exc) false;
@*/

public boolean addCategory(Category c) {
```
Normalisation: Example

- non_null by default

```java
/*@ public behavior
@ requires c.id >= 0;
@ requires c != null;
@ ensures \result == (...);
@ signals (Throwable exc) false;
@*/

public boolean addCategory(/*@ nullable @*/ Category c) {
```
Normalisation: Example

- Implicit `\invariant_for(this)` as requires, ensures & signals clause

```java
/*@ public behavior */
@ requires c.id >= 0;
@ requires c != null;
@ requires \invariant_for(this);
@ ensures \result == ( ... );
@ ensures \invariant_for(this);
@ signals (Throwable exc) false;
@ signals (Throwable exc) \invariant_for(this);
@*/

public /*@ helper @*/
    boolean addCategory(/*@ nullable @*/ Category c) {
```
Normalisation: Example

- Implicit specification explicit

```java
/*@ public behavior
  @ requires c.id >= 0;
  @ requires c != null;
  @ requires \text{invariant for}(this);
  @ ensures \text{result} == ( ... );
  @ ensures \text{invariant for}(this);
  @ signals (Throwable exc) false;
  @ signals (Throwable exc) \text{invariant for}(this);
  @*/

public /*@ helper */
  boolean addCategory(/*@ nullable */ Category c) {
```
Normalisation: Processing of Modifiers

Processing of Modifiers (Not detailed)

Expanding pure modifier: add to each specification case:

- assignable \nothing;
- diverges false;

Adding of default clauses if not present

Where clauses with defaults (e.g., diverges, assignable) are absent, add explicit clauses.
Clause Contraction

Merge multiple clauses of the same kind into a single one of that kind.

For instance,

```java
/*@ public behavior
 @ requires R1;
 @ requires R2;
 @ ensures E1;
 @ ensures E2;
 @ signals (T1 exc) S1;
 @ signals (T2 exc) S2:
 @*/
```
Clause Contraction

Merge multiple clauses of the same kind into a single one of that kind.

For instance,

```*/
/** public behavior
  @ requires R1;
  @ requires R2;
  @ ensures E1;
  @ ensures E2;
  @ signals (T1 exc) S1;
  @ signals (T2 exc) S2:
*/
```

```*/
/** public behavior
  @ requires R1 && R2;
  @ ensures E1 && E2;
  @ signals (Throwable exc)
    @ (exc instanceof T1 ==> S1)
    @ &&
    @ (exc instanceof T2 ==> S2);
*/
```
Intermediate format for contract of method $m$

$(pre, post, div, var, mod)$

with

- a precondition DL formula $pre$,
- a postcondition DL formula $post$,
- a divergence indicator $div \in \{ TOTAL, PARTIAL \}$,
- a variant $var$ a term of type any
- a modifies set $mod$, either of type LocSet or strictly_nothing
Translating JML Expressions to DL-Terms: Arithmetic Expressions

Translation replaces arithmetic JAVA operators by generalized operators
Generic towards various integer semantics (JAVA, Math).

Example:

“+” becomes “javaAddInt” or “javaAddLong”
“-” becomes “javaSubInt” or “javaSubLong”

...
Translating JML Expressions to DL-Terms: The this Reference

The this reference, explicit or implicit, has only a meaning within a program (refers to currently executing instance).

On logic level (outside the modalities) no such context exists.

The this reference translated to a program variable (named by convention) self
Translating JML Expressions to DL-Terms: The this Reference

The this reference, explicit or implicit, has only a meaning within a program (refers to currently executing instance).

On logic level (outside the modalities) no such context exists.

this reference translated to a program variable (named by convention) self

e.g., given class

public class MyClass {
    int f;
}

JML expressions f and this.f translated to
DL term select(heap, self, f)
Translating Boolean JML Expressions

First-order logic treated fundamentally different in JML and KeY logic

**JML**
- Formulas no separate syntactic category
- Instead: JAVA’s `boolean` expressions extended with first-order concepts (i.p. quantifiers)

**Dynamic Logic**
- Formulas and expressions completely separate
- Truth constants `true, false` are formulas, `boolean` constants `TRUE, FALSE` are terms
- Atomic formulas take terms as arguments; e.g.:
  - `x - y < 5`
  - `b = TRUE`
Translating Boolean JML Expressions

\[ F(v) = v = \text{TRUE} \]
\[ F(o.f) = E(o.f) = \text{TRUE} \]
\[ F(m()) = E(m)() = \text{TRUE} \]
\[ F(!b_0) = \neg F(b_0) \]
\[ F(b_0 \&\& b_1) = F(b_0) \& F(b_1) \]
\[ F(b_0 \mid\mid b_1) = F(b_0) | F(b_1) \]
\[ F(b_0 \implies b_1) = F(b_0) \rightarrow F(b_1) \]
\[ F(b_0 \iff b_1) = F(b_0) \iff F(b_1) \]
\[ F(e_0 == e_1) = E(e_0) = E(e_1) \]
\[ F(e_0 != e_1) = \neg E(e_0) = E(e_1) \]
\[ F(e_0 \geq e_1) = E(e_0) \geq E(e_1) \]

v/f/m() \textbf{boolean} variables/fields/pure methods
b_0, b_1 \textbf{boolean} JML expressions, e_0, e_1 JML expressions
E translates JML expressions to DL terms
\( \mathcal{F} \) Translates boolean JML Expressions to Formulas

Quantified formulas over reference types:

\[
\mathcal{F}(\forall T x; e_0; e_1) = \\
\forall T x; ( \\
\quad (!x=\text{null} \ \& \ \text{select(heap,x,\text{<created>})}=\text{TRUE} \ \& \ \mathcal{F}(e_0)) \\
\quad \rightarrow \mathcal{F}(e_1))
\]

\[
\mathcal{F}(\exists T x; e_0; e_1) = \\
\exists T x; ( \\
\quad (!x=\text{null} \ \& \ \text{select(heap,x,\text{<created>})}=\text{TRUE} \ \& \ \mathcal{F}(e_0)) \\
\quad \& \ \mathcal{F}(e_1))
\]
Translates boolean JML Expressions to Formulas

Quantified formulas over primitive types, e.g., int

\[ \mathcal{F}(∀ \text{int } x; \ e_0; \ e_1) = \exists \text{int } x; (\text{inInt}(x) \ & \ \mathcal{F}(e_0)) \rightarrow \mathcal{F}(e_1) \]

\[ \mathcal{F}(∃ \text{int } x; \ e_0; \ e_1) = \forall \text{int } x; (\text{inInt}(x) \ & \ \mathcal{F}(e_0) \ & \ \mathcal{F}(e_1)) \]

\text{inInt} (\text{similar } \text{inLong}, \text{inByte}): \]

Predefined predicate symbol with fixed interpretation

\textbf{Meaning}: Argument is within the range of the Java int datatype.
Translating Class Invariants

\[ \mathcal{F}(\text{invariant}\_\text{for}(e)) = \text{Object} :: <\text{inv}> (\text{heap}, \mathcal{E}(e)) \]

- \text{invariant}\_\text{for}(e) translated to built-in predicate \text{Object} :: <\text{inv}>, applied to \text{heap} and the translation of \( e \)
- \text{Object} :: <\text{inv}> is considered a specification-only field <\text{inv}> of class \text{Object} (inherited by all sub-types of \text{Object})
- Given that \( o \) is of type \( T \), KeY can expand \text{Object} :: <\text{inv}> (\text{heap}, o) to the invariant of \( T \)
- \text{Object} :: <\text{inv}> (\text{heap}, o) pretty printed as \( o.<\text{inv}>() \)
- Read ‘invariant of \( o \)’
Intermediate format for contract of method $m$

$(pre, post, div, var, mod)$

with

- a precondition DL formula $pre$ ✓,
- a postcondition DL formula $post$ ✓?
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,
- a variant $var$ a term of type any,
- a modifies set $mod$, either of type LocSet or \strictly_nothing
Intermediate format for contract of method $m$

$$(\text{pre}, \text{post}, \text{div}, \text{var}, \text{mod})$$

with

- a precondition DL formula $\text{pre}$,
- a postcondition DL formula $\text{post}$ almost,
- a divergence indicator $\text{div} \in \{\text{TOTAL}, \text{PARTIAL}\}$,
- a variant $\text{var}$ a term of type $\text{any}$,
- a modifies set $\text{mod}$, either of type $\text{LocSet}$ or $\text{strictly_nothing}$
Translation of Ensures Clauses

What is missing for ensures clauses?
Translation of EnsuresClauses

What is missing for ensures clauses?

- Translation of \result
- Translation of \old(.) expressions
Translation of Ensures Clauses

Translating \texttt{result}

For \texttt{result} used in ensures clause of method \texttt{T m(...)}:

$$\mathcal{E}(\texttt{result}) = \texttt{result}$$

where \texttt{result} ∈ \textit{PVar} of type \texttt{T} does not occur in the program.
Translating \texttt{old} Expressions

\texttt{old}(e) evaluates \(e\) in the prestate of the method.
Accesses to \texttt{heap} must be evaluated w.r.t. the 'old' heap.
Translating \texttt{old} Expressions

\texttt{old}(e) evaluates $e$ in the prestate of the method

Accesses to \texttt{heap} must be evaluated w.r.t. the 'old' heap

1. Introduce a global program variables $\texttt{heapAtPre}$ of type \texttt{Heap}
   (Intention: $\texttt{heapAtPre}$ refers to heap in method's pre-state)

2. Define:
   \[ \mathcal{E}(\texttt{old}(e)) = \mathcal{E}_{\texttt{heapAtPre}}(e) \]
   \[ (\mathcal{E}_x^y(e) \text{ replaces all occurrences of } x \text{ in } \mathcal{E}(e) \text{ by } y) \]
Translating `\texttt{old}` Expressions

`\texttt{old}(e)` evaluates `e` in the prestate of the method.

Accesses to `heap` must be evaluated w.r.t. the 'old' heap.

1. Introduce a global program variables `heapAtPre` of type `Heap` (Intention: `heapAtPre` refers to heap in method’s pre-state)

2. Define:
   \[
   \mathcal{E}(\texttt{old}(e)) = \mathcal{E}_{\texttt{heapAtPre}}(e)
   \]
   \[
   (\mathcal{E}_{x'y}(e) \text{ replaces all occurrences of } x \text{ in } \mathcal{E}(e) \text{ by } y)
   \]

Example

\[
\mathcal{F}(o.f == \texttt{old}(o.f) + 1) =
\]
Translating `\old` Expressions

`\old(e)` evaluates `e` in the prestate of the method
Accesses to `heap` must be evaluated w.r.t. the 'old' heap

1. Introduce a global program variables `heapAtPre` of type `Heap`
   (Intention: `heapAtPre` refers to heap in method’s pre-state)

2. Define:
   \[
   \mathcal{E}(\old(e)) = \mathcal{E}_{\text{heap}}^{\text{heapAtPre}}(e)
   \]
   \[
   (\mathcal{E}^y_x(e) \text{ replaces all occurrences of } x \text{ in } \mathcal{E}(e) \text{ by } y)
   \]

Example

\[
\mathcal{F}(o.f \text{ == } \old(o.f)+ 1) = \mathcal{E}(o.f) = \mathcal{E}(\old(o.f)+ 1) =
\]
Translating \texttt{\textbackslash old} Expressions

\texttt{\textbackslash old}(e) evaluates \emph{e} in the prestate of the method

Accesses to heap must be evaluated w.r.t. the 'old' heap

1. Introduce a global program variables \texttt{heapAtPre} of type Heap
   (Intention: \texttt{heapAtPre} refers to heap in method’s pre-state)

2. Define:
   \[ E(\texttt{\textbackslash old}(e)) = E_{\texttt{heapAtPre}}(e) \]
   \[ (E_X^Y(e) \text{ replaces all occurrences of } x \text{ in } E(e) \text{ by } y) \]

Example

\[ F(o.f == \texttt{\textbackslash old}(o.f)+ 1) = \]
\[ E(o.f) = E(\texttt{\textbackslash old}(o.f)+ 1) = \]
\[ E(o.f) = E(\texttt{\textbackslash old}(o.f)) + E(1) = \]
Translating \texttt{old} Expressions

\texttt{old}(e) evaluates e in the prestate of the method

Accesses to heap must be evaluated w.r.t. the 'old' heap

1. Introduce a global program variables \texttt{heapAtPre} of type Heap
   (Intention: \texttt{heapAtPre} refers to heap in method’s pre-state)

2. Define:

   \[ E(\texttt{old}(e)) = E_{\text{heapAtPre}}(e) \]

   \( (E^y_x(e) \text{ replaces all occurrences of } x \text{ in } E(e) \text{ by } y) \)

Example

\[ F(o.f == \texttt{old}(o.f)+ 1) = \]

\[ E(o.f) = E(\texttt{old}(o.f)+ 1) = \]

\[ E(o.f) = E(\texttt{old}(o.f)) + E(1) = \]

\[ E(o.f) = E_{\text{heapAtPre}}(o.f) + 1 = \]
Translating \old Expressions

\old(e) evaluates e in the prestate of the method
Accesses to heap must be evaluated w.r.t. the 'old' heap

1. Introduce a global program variables heapAtPre of type Heap
   (Intention: heapAtPre refers to heap in method’s pre-state)

2. Define:
   \[ E(\old(e)) = E^{\text{heapAtPre}}_{\text{heap}}(e) \]
   \[ (E^y_x(e) \text{ replaces all occurrences of } x \text{ in } E(e) \text{ by } y) \]

Example

\[ F(o.f == \old(o.f)+ 1) = \]
\[ E(o.f) = E(\old(o.f)+ 1) = \]
\[ E(o.f) = E(\old(o.f)) + E(1) = \]
\[ E(o.f) = E^{\text{heapAtPre}}_{\text{heap}}(o.f) + 1 = \]
\[ \text{select(heap, o, f)} = \text{select(heapAtPre, o, f)} + 1 \]
Translation of Ensures and Signals Clauses

Given the normalised JML contract

/*@ public behavior @*
  @ ... @
  @ ensures E;
  @ signals (Throwable exc) S;
  @ ...
  @*/
Translation of Ensures and Signals Clauses

Given the normalised JML contract

```java
/*@ public behavior
   @ ...
   @ ensures E;
   @ signals (Throwable exc) S;
   @ ...
   @*/
```

Define

\[ \mathcal{F}_{\text{ensures}} = \mathcal{F}(E) \]
\[ \mathcal{F}_{\text{signals}} = \mathcal{F}(S) \]
Translation of Ensures and Signals Clauses

Given the normalised JML contract

```java
/*@ public behavior 
  @ ... 
  @ ensures E; 
  @ signals (Throwable exc) S; 
  @ ... 
  @*/
```

Define

\[ F_{\text{ensures}} = F(E) \]
\[ F_{\text{signals}} = F(S) \]

Recall that \( S \) is either false, or it has the form

\[ (\text{exc instanceof ExcType1} \implies \text{ExcPost1}) \land ...; \]

In the following, assume \text{exc} is fresh program variable of type Throwable
Combining Signals and Ensures to $post$

The DL formula $post$ is then defined as

$$( \text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}} ) \& ( \text{exc}! = \text{null} \rightarrow \mathcal{F}_{\text{signals}} )$$
Combining Signals and Ensures to \textit{post}

The DL formula \textit{post} is then defined as

\[(\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \land (\text{exc}! = \text{null} \rightarrow \mathcal{F}_{\text{signals}})\]

\textbf{Note:}

Normalisation of \texttt{normal\_behavior} contract gives \texttt{signals} (\texttt{Throwable exc}) \texttt{false};

Then \textit{post} is:

\[(\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \land (\text{exc}! = \text{null} \rightarrow \mathcal{F}(\text{false}))\]

\[\Leftrightarrow (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \land (\text{exc}! = \text{null} \rightarrow \text{false})\]

\[\Leftrightarrow (\text{exc} = \text{null} \rightarrow \mathcal{F}_{\text{ensures}}) \land \text{exc} = \text{null}\]

\[\Leftrightarrow \text{exc} = \text{null} \land \mathcal{F}_{\text{ensures}}\]
# Translating JML into Intermediate Format

## Intermediate format for contract of method $m$

\[(pre, post, div, var, mod)\]

with

- a precondition DL formula $pre$,
- a postcondition DL formula $post$,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,
- a variant $var$ a term of type any,
- a modifies set $mod$, either of type LocSet or \texttt{\textbackslash strictly\_nothing}
The Divergence Indicator

\[
div = \begin{cases} 
    TOTAL & \text{if normalised JML contract contains clause diverges \ false;} \\
    PARTIAL & \text{if normalised JML contract contains clause diverges \ true;}
\end{cases}
\]
### Intermediate format for contract of method \( m \)

\[(pre, post, div, var, mod)\]

with
- a precondition DL formula \( pre \),
- a postcondition DL formula \( post \),
- a divergence indicator \( div \in \{ TOTAL, PARTIAL \} \),
- a variant \( var \) a term of type any,
- a modifies set \( mod \), either of type LocSet or \strictly\_nothing\,
Intermediate format for contract of method \( m \)

\[(pre, post, div, var, mod)\]

with

- a precondition DL formula \( pre \),
- a postcondition DL formula \( post \),
- a divergence indicator \( div \in \{TOTAL, PARTIAL\} \),
- a variant \( var \) a term of type \( \text{any} \) (postponed to later lecture),
- a modifies set \( mod \), either of type \( \text{LocSet} \) or \( \text{\textbackslash strictly\_nothing} \)
Assignable clauses are translated to

a term of type LocSet or the special value $\text{\textbackslash strictly\_nothing}$
Assign able clauses are translated to

a term of type \texttt{LocSet} or the special value \texttt{strictly\_nothing}

\textbf{Intention:} A term of type \texttt{LocSet} represents a set of locations

\textbf{Definition (Locations)}

A location is a tuple \((o, f)\) with \(o \in D^{\text{Object}}, f \in D^{\text{Field}}\)

\textbf{Note:} Location is a \textit{semantic} and not a syntactic entity.
The DL Type LocSet

Predefined type with $D(LocSet) = 2^{Location}$ and the functions (all with result type LocSet):

- **empty**
  - empty set of locations: $I(\text{empty}) = \emptyset$

- **allLocs**
  - set of all locations, i.e., $I(\text{allLocs}) = \{(d, f) | f.a. d \in D^{Object}, f \in D^{Field}\}$

- **singleton(Object, Field)**
  - singleton set

- **union(LocSet, LocSet)**

- **intersect(LocSet, LocSet)**

- **allFields(Object)**
  - set of all locations for the given object; e.g., $\{(d, f) | f.a. d \in D^{Object}\}$

- **allObjects(Field)**

- **arrayRange(Object, int, int)**
  - set representing all array locations in the specified range (both inclusive)
Example

assignable \texttt{\textbackslash everything};

is translated into the DL term
Example

assignable \texttt{\textbackslash everything};
is translated into the DL term

\texttt{allLocs}
Translating Assignable Clauses—Example

Example

assignable \ everything;
is translated into the DL term

allLocs

Example

assignable this.next, this.content[5..9];
is translated into the DL term
Example

assignable \texttt{\textbackslash everything};

is translated into the DL term

\texttt{allLocs}

Example

assignable \texttt{this.next, this.content[5..9]};

is translated into the DL term

\texttt{union(singleton(self, next),
arrayRange(select(heap, self, context), 5, 9))}
Translating JML into Intermediate Format

Intermediate format for contract of method \( m \)

\[
(pre, post, div, var, mod)
\]

with

- a precondition DL formula \( pre \)
- a postcondition DL formula \( post \)
- a divergence indicator \( div \in \{TOTAL, PARTIAL\} \)
- a variant \( var \) a term of type \( \text{any} \) (postponed)
- a modifies set \( mod \), either of type \( \text{LocSet} \) or \( \text{strictly_nothing} \)
public class A {
    /*@ public normal_behavior
        @ requires <Precondition>;
        @ ensures <Postcondition>;
        @ assignable <locations>;
        @*/
    public int m(params) { .. }
}
Generating a PO from the Intermediate Format: Idea

Given intermediate format of contract of \( m \) implemented in class \( C \):

\[
\left( pre, \ post, \ TOTAL, \ var, \ mod \right)
\]

\[
pre \rightarrow \langle self.m(args) \rangle (post \ & \ \text{frame})
\]

\text{correctness of assignable}
Generating a PO from the Intermediate Format: Idea

Given intermediate format of contract of \( m \) implemented in class \( C \):

\[
(pre, post, TOTAL, var, mod)
\]

\[
pre \rightarrow \langle\text{self.m(args)}\rangle (post \ & \ \text{frame} )
\]

(correctness of assignable)

(In case of \( \text{div} = \text{PARTIAL} \), box modality is used)
Generating a PO from Intermediate Format: Method Identification

\[ pre \rightarrow \langle \text{self.m(args)} \rangle (post \ & \ frame) \]
Generating a PO from Intermediate Format: Method Identification

\[ pre \to \langle \text{self.m(args)} \rangle (post \ & \ frame) \]

- Dynamic dispatch: \texttt{self.m(...)} causes split into all possible implementations
Generating a PO from Intermediate Format:
Method Identification

\[ pre \rightarrow \langle \text{self.m(args)} \rangle (post \ & \ frame) \]

- Dynamic dispatch: \texttt{self.m(...)} causes split into all possible implementations
- Special statement \textbf{Method Body Statement}:

\[ \text{m(args)}@C \]

Meaning: Placeholder for the method body of class \( C \)
Generating a PO from Intermediate Format: Exceptions

\[ pre \rightarrow \langle \text{self.m(args)} \rangle^C (post \ & \ frame) \]

Postcondition \emph{post} states either

\begin{itemize}
  \item that no exception is thrown or
  \item that in case of an exception the exceptional postcondition holds
\end{itemize}

but: \( \langle \text{throw } \text{exc}; \rangle \varphi \) is trivially false

How to refer to an exception in post-state?
Generating a PO from Intermediate Format: Exceptions

\[ pre \rightarrow \langle \text{self.m(args)} @ C \rangle (post \ & \ frame) \]

Postcondition \( post \) states either

- that no exception is thrown or
- that in case of an exception the exceptional postcondition holds

but: \( \langle \text{throw exc;} \rangle \varphi \) is trivially false

How to refer to an exception in post-state?

\[ pre \rightarrow \]

\[
\langle \begin{cases} 
\text{exc} = \text{null;} \\
\text{try} \left\{ \\
\text{self.m(args)} @ C \\
\text{catch} \ (\text{Throwable t})\{\text{exc} = t;\} \\
\end{cases} \rangle (post \ & \ frame)
\]

(Recall: Normalisation and post-generation used program variable \( \text{exc} \))
The Generic Precondition $\textit{genPre}$

$$\textit{pre} \rightarrow \langle \text{exc=\texttt{null}; try \{self.m(args)@C\} catch ...} \rangle (\textit{post \& frame})$$

is still not complete.
The Generic Precondition \textit{genPre}

\[
pre \rightarrow \langle \text{exc=\texttt{null}; try \{self.m(args)@C\} catch ...} \rangle (post \ & \ frame)
\]

is still not complete.

Additional properties (known to hold in Java, but not in DL), e.g.,

\begin{itemize}
  \item this is not null
  \item created objects can only point to created objects (no dangling references)
  \item integer parameters have correct range
  \item ...
\end{itemize}
The Generic Precondition \( \text{genPre} \)

\[
\pre \rightarrow \langle \text{exc=}\text{null}; \ \text{try} \ \{\text{self.m(args)}\@C\} \ \text{catch} \ \ldots \ \rangle \ (\text{post} \ \& \ \text{frame})
\]

is still not complete.

Additional properties (known to hold in Java, but not in DL), e.g.,

- this is not null
- created objects can only point to created objects (no dangling references)
- integer parameters have correct range
- \ldots

Need to make these assumption on initial state explicit in DL.

**Idea:** Formalise assumption as additional precondition \( \text{genPre} \)

\[
(\text{genPre} \ \& \ \pre) \rightarrow \\
\langle \text{exc=}\text{null}; \ \text{try} \ \{\text{self.m(args)}\@C\} \ \text{catch} \ \ldots \ \rangle \ (\text{post} \ \& \ \text{frame})
\]
The Generic Precondition \( \text{genPre} \)

\[
\text{genPre} := \text{wellFormed}(\text{heap}) \\
\land \text{paramsInRange} \\
\land \text{self} \neq \text{null} \\
\land \text{boolean} :: \text{select}(\text{heap}, \text{self}, <\text{created}>) = \text{TRUE} \\
\land \text{C} :: \text{exactInstance}(\text{self}) \\
\land \text{exc} = \text{null}
\]

- **wellFormed**: predefined predicate; true iff. given heap is regular Java heap
- **paramsInRange** formula stating that the method arguments are in range
- **C :: exactInstance**: predefined predicate; true iff. given argument has C as exact type (i.e., is not of a subtype)
The Generic Precondition \( genPre \)

\[
(genPre \land pre) \rightarrow \langle \text{exc=}null; \text{try \{self.m(args)@C\} catch ...} \rangle (post \land frame)
\]

is still not complete.

- Need to refer to prestate in post, e.g. for old-expressions
The Generic Precondition \textit{genPre}

\[(\text{genPre} \land \text{pre}) \rightarrow \langle \text{exc=} \text{null}; \text{try} \{\text{self.m(args)}@C\} \text{ catch ... } \rangle (\text{post} \land \text{frame})\]

is still not complete.

- Need to refer to prestate in post, e.g. for old-expressions

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre := heap}\}
\langle \text{exc=} \text{null}; \text{try} \{\text{self.m(args)}@C\} \text{ catch ... } \rangle (\text{post} \land \text{frame})\]

(Reminder: \textit{heapAtPre} was used in translation of \textbackslash old in \textit{post})
Generating a PO from Intermediate Format: Result Value

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre} := \text{heap}\}\]

\[\langle \text{exc} = \text{null}; \text{try} \{\text{self.m(args)}\} \text{catch} \ldots \rangle\]

\[(\text{post} \land \text{frame})\]

is still not complete.

- For non-void methods, need to refer to result in \textit{post}
Generating a PO from Intermediate Format: Result Value

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre} := \text{heap}\}\]
\[
  \langle \text{exc=\text{null}; try \{self.m(args)\} catch \ldots } \rangle
  \]
\[
  (\text{post \& frame})
\]
is still not complete.

- For non-void methods, need to refer to result in \textit{post}

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre} := \text{heap}\}\]
\[
  \langle \text{exc=\text{null}; try \{\text{result} = self.m(args)\} catch \ldots } \rangle
  \]
\[
  (\text{post \& frame})
\]

(Reminder: result was used in translation of \texttt{\textbackslash result} in \textit{post})
Generating a PO from Intermediate Format:
The frame DL Formula

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre} := \text{heap}\}\]
\[\langle \text{exc=}\text{null}; \text{try} \{\text{self.m(args)}\} \text{ catch ...} \rangle\]
\[(\text{post} \& \text{ frame})\]

If \(mod = \text{strictly_nothing}\) then \text{frame} is defined

\[\forall o; \forall f; (\text{select(\text{heapAtPre}, o, f)} = \text{select(\text{heap}, o, f)})\]
Generating a PO from Intermediate Format: The \textit{frame} DL Formula

\[(\text{genPre} \land \text{pre}) \rightarrow \{\text{heapAtPre} := \text{heap}\} \]
\[
\langle \text{exc=}\text{null}; \text{try} \{\text{self.m(args)}\} \text{ catch } \ldots \rangle
\]
\[
(\text{post} \land \text{frame})
\]

If \textit{mod} is a location set, then \textit{frame} is defined as:

\[
\forall o; \forall f; \left( \text{select}(\text{heapAtPre}, o, \langle \text{created} \rangle) = \text{FALSE} \right.
\]
\[
\lor \text{select}(\text{heapAtPre}, o, f) = \text{select}(\text{heap}, o, f)
\]
\[
\lor (o, f) \in \{\text{heap} := \text{heapAtPre}\} \text{mod}
\]
Generating a PO from Intermediate Format: 
The frame DL Formula

\[(genPre \land pre) \rightarrow \{heapAtPre := heap\}\]

\[
\langle \text{exc}=\text{null}; \ \text{try} \ \{\text{self.m(args)}\} \ \text{catch} \ \ldots \ \rangle
\]

(post \& frame)

If \(mod\) is a location set, then frame is defined as:

\[\forall o; \forall f; (\ select(\text{heapAtPre}, o, \langle \text{created}\rangle) = \text{FALSE} \]
\[
\lor select(\text{heapAtPre}, o, f) = select(\text{heap}, o, f) \]
\[
\lor (o, f) \in \{heap := heapAtPre\} mod
\]

States that any location \((o, f)\)

- belongs to an object was not (yet) created before the method invocation, or
- holds the same value after the invocation as before the invocation, or
- belongs to the modifies set (evaluated in the pre-state).
Examples

Demo
Literature for this Lecture

Essential

KeY Quicktour see course page, under 'Links, Papers, and Software'