# Formal Methods for Software Development

## Proof Obligations

Wolfgang Ahrendt

18 October 2019

# This Part

making the connection between

JML

and

Dynamic Logic / KeY

# This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,

# This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,

# This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,
- and proving

DL proof obligations from JML specifications

# From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

# From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

**Translation**

Intermediate Format

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

# From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

**Translation**

Intermediate Format

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

**PO Generation**

Proof obligation as DL formula

$$pre \rightarrow \langle \texttt{this.m(params);} \rangle (post \ \wedge \ frame)$$

# 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

Tho following introduces principles of this process

# Normalisation: Making Implicit Information Explicit

**Implicit Information**

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

# Normalisation:
# Making Implicit Information Explicit

**Implicit Information**

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

**Turn into general `behavior` spec. case**

1. Add to
   - `normal_behavior` the clause **`signals (Throwable t) false;`**

# Normalisation:
# Making Implicit Information Explicit

## Implicit Information

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

## Turn into general `behavior` spec. case

1. Add to
   - `normal_behavior` the clause **`signals (Throwable t) false;`**
   - `exceptional_behavior` the clause **`ensures false;`**

# Normalisation: Making Implicit Information Explicit

## Implicit Information

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

## Turn into general `behavior` spec. case

1. Add to
   - `normal_behavior` the clause **`signals (Throwable t) false;`**
   - `exceptional_behavior` the clause **`ensures false;`**
2. Replace `normal_behavior`/`exceptional_behavior` by `behavior`

# Normalisation: Making Implicit Information Explicit

## Implicit Information

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

## Making `non_null` explicit in method specifications

1. Where `nullable` is absent, add **`o != null`** to preconditions (for parameters[a]) and postconditions (for return values[a]).
   E.g., for method `void m(Object o)` add **`requires o != null;`**
2. Thereafter add **`nullable`**, where absent, to *all* parameter[a] and return type[a] declarations

[a] of reference type

# Normalisation: Making Implicit Information Explicit

### Implicit Information

- Meaning of `normal_` and `exceptional_behavior`
- `non_null` by default
- `\invariant_for(this)` in `requires`, `ensures`, `signals` clauses

### Making `\invariant_for(this)` explicit in method specifications

1. Add explicit `\invariant_for(this)` to non-helper method specs:
   - `requires \invariant_for(this);`
   - `ensures \invariant_for(this);`
   - `signals (Throwable t) \invariant_for(this);`
2. Thereafter add **`helper`**, where absent, to *all* methods

# Normalisation: Example

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

becomes

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

# Normalisation: Example

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

becomes

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

# Normalisation: Example

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

becomes

```
/*@ 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

## Next Normalisation Steps (Not detailed)

- Expanding `pure` modifier:
  - add to each specification case
    - **`assignable \nothing;`**
    - **`diverges false;`**
  - remove `pure`
- Where clauses with defaults (e.g., `diverges`, `assignable`) are absent, add explicit clauses

# Normalisation: 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:
  @*/
```

# Normalisation: 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);
  @*/
```

# 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 term $var$
- 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.

**`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;
}
```

# 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)`, pretty-printed as `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
- `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

$$
\begin{array}{lcl}
\mathcal{F}(\texttt{v}) & = & \texttt{v = TRUE} \\
\mathcal{F}(\texttt{o.f}) & = & \mathcal{E}(\texttt{o.f})\ \texttt{= TRUE} \\
\mathcal{F}(\texttt{m()}) & = & \mathcal{E}(\texttt{m})\texttt{()}\ \texttt{= TRUE} \\
\mathcal{F}(\texttt{!b\_0}) & = & \texttt{!}\mathcal{F}(\texttt{b\_0}) \\
\mathcal{F}(\texttt{b\_0 \&\& b\_1}) & = & \mathcal{F}(\texttt{b\_0})\ \texttt{\&}\ \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 || b\_1}) & = & \mathcal{F}(\texttt{b\_0})\ \texttt{|}\ \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 ==> b\_1}) & = & \mathcal{F}(\texttt{b\_0})\ \texttt{->}\ \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{b\_0 <==> b\_1}) & = & \mathcal{F}(\texttt{b\_0})\ \texttt{<->}\ \mathcal{F}(\texttt{b\_1}) \\
\mathcal{F}(\texttt{e\_0 == e\_1}) & = & \mathcal{E}(\texttt{e\_0})\ \texttt{=}\ \mathcal{E}(\texttt{e\_1}) \\
\mathcal{F}(\texttt{e\_0 != e\_1}) & = & \texttt{!(}\mathcal{E}(\texttt{e\_0})\ \texttt{=}\ \mathcal{E}(\texttt{e\_1})\texttt{)} \\
\mathcal{F}(\texttt{e\_0 >= e\_1}) & = & \mathcal{E}(\texttt{e\_0})\ \texttt{>=}\ \mathcal{E}(\texttt{e\_1})
\end{array}
$$

`v`/`f`/`m()` **`boolean`** variables/fields/pure methods

`b_0`, `b_1` **`boolean`** JML expressions, `e_0`, `e_1` JML expressions

$\mathcal{E}$ translates JML expressions to DL terms

# $\mathcal{F}$ Translates `boolean` JML Expressions to Formulas

Quantified formulas over reference types:

```
F((\forall T x; e_0; e_1)) =
 \forall T x; (
      (!x=null  &  x.<created> = TRUE  &  F(e_0))
   -> F(e_1))

F((\exists T x; e_0; e_1)) =
 \exists T x; (
      (!x=null  &  x.<created> = TRUE  &  F(e_0))
   &  F(e_1))
```

# $\mathcal{F}$ Translates `boolean` JML Expressions to Formulas

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

$\mathcal{F}$((**\forall** int x; e_0; e_1)) =
    **\forall** int x; ((inInt(x) & $\mathcal{F}$(e_0)) -> $\mathcal{F}$(e_1))

$\mathcal{F}$((**\exists** int x; e_0; e_1)) =
    **\exists** int x; (inInt(x) & $\mathcal{F}$(e_0) & $\mathcal{F}$(e_1))

inInt (similar inLong, inByte):

Predefined predicate symbol with fixed interpretation

**Meaning:** Argument is within the range of the Java **int** datatype.

# Translating Class Invariants

$$\mathcal{F}(\texttt{\textbackslash invariant\_for(e)}) \quad = \quad \texttt{Object}::\!<\!\texttt{inv}\!>(\texttt{heap}, \mathcal{E}(\texttt{e}))$$

- \invariant_for(e) translated to built-in predicate $\texttt{Object}::<\texttt{inv}>$, applied to heap and the translation of e

# Translating Class Invariants

$$\mathcal{F}(\texttt{\textbackslash invariant\_for}(\texttt{e})) \quad = \quad \texttt{Object}::\!<\!\texttt{inv}\!>(\texttt{heap}, \mathcal{E}(\texttt{e}))$$

- \invariant_for(e) translated to built-in predicate $\texttt{Object}::\!<\!\texttt{inv}\!>$, applied to heap and the translation of e
- $\texttt{Object}::\!<\!\texttt{inv}\!>$ is considered a specification-only field $<\!\texttt{inv}\!>$ of class Object (inherited by all sub-types of Object)

# Translating Class Invariants

$$\mathcal{F}(\texttt{\textbackslash invariant\_for}(\texttt{e})) \quad = \quad \texttt{Object}::\texttt{<inv>}(\texttt{heap}, \mathcal{E}(\texttt{e}))$$

- `\invariant_for(e)` translated to built-in predicate $\texttt{Object}::\texttt{<inv>}$, applied to `heap` and the translation of `e`
- $\texttt{Object}::\texttt{<inv>}$ is considered a specification-only field $\texttt{<inv>}$ of class `Object` (inherited by all sub-types of `Object`)
- Given that $o$ is of type $T$, KeY can expand (during proof construction) '$\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$' to the invariant of $T$

# Translating Class Invariants

$$\mathcal{F}(\texttt{\textbackslash invariant\_for(e)}) \quad = \quad \texttt{Object}::\texttt{<inv>}(\texttt{heap}, \mathcal{E}(\texttt{e}))$$

- `\invariant_for(e)` translated to built-in predicate $\texttt{Object}::\texttt{<inv>}$, applied to `heap` and the translation of `e`
- $\texttt{Object}::\texttt{<inv>}$ is considered a specification-only field $\texttt{<inv>}$ of class `Object` (inherited by all sub-types of `Object`)
- Given that $o$ is of type $T$, KeY can expand (during proof construction) '$\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$' to the invariant of $T$
- $\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$ pretty printed as $o.\texttt{<inv>}$

# Translating Class Invariants

$$\mathcal{F}(\texttt{\textbackslash invariant\_for(e)}) \quad = \quad \texttt{Object}::\texttt{<inv>}(\texttt{heap}, \mathcal{E}(\texttt{e}))$$

- `\invariant_for(e)` translated to built-in predicate $\texttt{Object}::\texttt{<inv>}$, applied to `heap` and the translation of `e`
- $\texttt{Object}::\texttt{<inv>}$ is considered a specification-only field $\texttt{<inv>}$ of class `Object` (inherited by all sub-types of `Object`)
- Given that $o$ is of type $T$, KeY can expand (during proof construction) '$\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$' to the invariant of $T$
- $\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$ pretty printed as $o.\texttt{<inv>}$
- Read 'invariant of $o$'

# 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 term *var*
- a modifies set *mod*, either of type `LocSet` or `\strictly_nothing`

# 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 $post$ ✔almost,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$,
- a variant term $var$
- a modifies set $mod$, either of type `LocSet` or `\strictly_nothing`

# Translation of Ensures Clauses

What is missing for `ensures` clauses?

# Translation of Ensures Clauses

What is missing for `ensures` clauses?

- Translation of `\result`
- Translation of `\old(.)` expressions

# Translation of Ensures Clauses

What is missing for `ensures` clauses?

- Translation of `\result`
- Translation of `\old(.)` expressions

**Translating `\result`**

For `\result` used in ensures clause of method $T\ m(\ldots)$:

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

where $\texttt{result} \in \mathit{PVar}$ of type $T$ does not occur in the program.

# Translating `\old` Expressions

`\old`($e$) evaluates $e$ in the prestate of the method

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

# Translating `\old` Expressions

`\old`($e$) evaluates $e$ in the prestate of the method

Accesses to `heap` must be evaluated w.r.t. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}_{\texttt{heap}}^{\texttt{heapAtPre}}(e)$

   ($\mathcal{E}_x^y(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

# Translating `\old` Expressions

`\old`($e$) evaluates $e$ in the prestate of the method

Accesses to `heap` must be evaluated w.r.t. to 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:

   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(e)$

   ($\mathcal{E}^{y}_{x}(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

**Example**

$\mathcal{F}($`o.f == \old(o.f)+ 1`$) \quad =$

# Translating `\old` Expressions

`\old`($e$) evaluates $e$ in the prestate of the method

Accesses to `heap` must be evaluated w.r.t. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(e)$
   ($\mathcal{E}^y_x(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

**Example**

$\mathcal{F}$(`o.f == \old(o.f)+ 1`) $=$
$\mathcal{E}$(`o.f`) = $\mathcal{E}$(`\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. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(e)$
   ($\mathcal{E}^{y}_{x}(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

**Example**

$\mathcal{F}(\texttt{o.f == \textbackslash old(o.f)+ 1}) \quad =$
$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)+ 1}) \quad =$
$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)}) \texttt{ + } \mathcal{E}(\texttt{1}) \quad =$

# Translating \old Expressions

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

Accesses to heap must be evaluated w.r.t. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}_{\texttt{heap}}^{\texttt{heapAtPre}}(e)$
   ($\mathcal{E}_x^y(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

**Example**

$\mathcal{F}(\texttt{o.f == \textbackslash old(o.f)+ 1}) \quad =$
$\mathcal{E}(\texttt{o.f}) = \mathcal{E}(\texttt{\textbackslash old(o.f)+ 1}) \quad =$
$\mathcal{E}(\texttt{o.f}) = \mathcal{E}(\texttt{\textbackslash old(o.f)}) + \mathcal{E}(1) \quad =$
$\mathcal{E}(\texttt{o.f}) = \mathcal{E}_{\texttt{heap}}^{\texttt{heapAtPre}}(\texttt{o.f}) + 1 \quad =$

# Translating `\old` Expressions

`\old`($e$) evaluates $e$ in the prestate of the method

Accesses to `heap` must be evaluated w.r.t. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(e)$

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

**Example**

$\mathcal{F}(\texttt{o.f == \textbackslash old(o.f)+ 1}) \quad =$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)+ 1}) \quad =$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)}) \texttt{ + } \mathcal{E}(\texttt{1}) \quad =$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(\texttt{o.f}) \texttt{ + 1} \quad =$

`select(heap, o, f) = select(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. to 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:
   $\mathcal{E}(\texttt{\textbackslash old}(e)) = \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(e)$

   ($\mathcal{E}^{y}_{x}(e)$ replaces all occurrences of $x$ in $\mathcal{E}(e)$ by $y$)

**Example**

$\mathcal{F}(\texttt{o.f == \textbackslash old(o.f)+ 1}) \;=$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)+ 1}) \;=$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}(\texttt{\textbackslash old(o.f)}) \texttt{ + } \mathcal{E}(\texttt{1}) \;=$

$\mathcal{E}(\texttt{o.f}) \texttt{ = } \mathcal{E}^{\texttt{heapAtPre}}_{\texttt{heap}}(\texttt{o.f}) \texttt{ + 1} \;=$

`select(heap, o, f) = select(heapAtPre, o, f) + 1` $=$

`o.f = o.f@heapAtPre + 1` *(by pretty printing)*

# 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

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

Define

$\mathcal{F}_{\text{ensures}} = \mathcal{F}(\mathtt{E})$

$\mathcal{F}_{\text{signals}} = \mathcal{F}(\mathtt{S})$

# Translation of Ensures and Signals Clauses

Given the normalised JML contract

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

Define

$\mathcal{F}_{\text{ensures}} = \mathcal{F}(\mathtt{E})$

$\mathcal{F}_{\text{signals}} = \mathcal{F}(\mathtt{S})$

Recall (pp.7,11) that `S` is either `false`, or it has the form

```
(exc instanceof ExcType1 ==> ExcPost1) && ...;
```

In the following, assume `exc` is fresh program variable of type `Throwable`

# Combining Ensures and Signals to *post*

The DL formula *post* is then defined as

$$(\texttt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\textsf{ensures}}) \ \wedge \ (\texttt{exc} \neq \mathbf{null} \rightarrow \mathcal{F}_{\textsf{signals}})$$

# Combining Ensures and Signals to *post*

The DL formula *post* is then defined as

$$(\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \wedge \ (\mathtt{exc} \neq \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{signals}})$$

**Important special case:**

Normalisation of `normal_behavior` contract gives

```
signals (Throwable exc) false;
```

# Combining Ensures and Signals to *post*

The DL formula *post* is then defined as

$$(\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \land \ (\mathtt{exc} \neq \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{signals}})$$

**Important special case:**

Normalisation of `normal_behavior` contract gives

`signals (Throwable exc) false;`

In that case, *post* is:

$$\begin{array}{ll} & (\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \land \ (\mathtt{exc} \neq \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{signals}}) \\ \Leftrightarrow & (\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \land \ (\mathtt{exc} \neq \mathbf{null} \rightarrow \mathcal{F}(\mathtt{false})) \\ \Leftrightarrow & (\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \land \ (\mathtt{exc} \neq \mathbf{null} \rightarrow \mathtt{false}) \\ \Leftrightarrow & (\mathtt{exc} = \mathbf{null} \rightarrow \mathcal{F}_{\mathsf{ensures}}) \ \land \ \mathtt{exc} = \mathbf{null} \\ \Leftrightarrow & \mathtt{exc} = \mathbf{null} \ \land \ \mathcal{F}_{\mathsf{ensures}} \end{array}$$

# 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 term $var$ ,
- a modifies set $mod$, either of type `LocSet` or `\strictly_nothing`

# 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 term $var$ ,
- a modifies set $mod$, either of type `LocSet` or `\strictly_nothing`

**The Divergence Indicator**

$$div = \begin{cases} TOTAL & \text{if normalised JML contract contains clause } \texttt{diverges}\ \mathbf{false}; \\ PARTIAL & \text{if normalised JML contract contains clause } \texttt{diverges}\ \mathbf{true}; \end{cases}$$

# 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 term $var$ ,
- a modifies set $mod$, either of type `LocSet` or `\strictly_nothing`

# 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 term *var* (postponed to later lecture),
- a modifies set *mod*, either of type `LocSet` or `\strictly_nothing`

# Translating Assignable Clauses: The DL Type `LocSet`

Assignable clauses are translated to

> a term of type `LocSet` or the special value `\strictly_nothing`

# Translating Assignable Clauses: The DL Type `LocSet`

Assignable clauses are translated to

a term of type `LocSet` or the special value `\strictly_nothing`

Intention: A term of type `LocSet` represents a set of locations

**Definition (Locations)**

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

# The DL Type LocSet

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

| | |
|---|---|
| empty | empty set of locations: $\mathcal{I}(\texttt{empty}) = \emptyset$ |
| allLocs | set of all locations, i.e., $\mathcal{I}(\texttt{allLocs}) = \{(d, f) \mid f.a.\ d \in D^{\texttt{Object}}, f \in D^{\texttt{Field}}\}$ |
| singleton(Object, Field) | singleton set |
| union(LocSet, LocSet) | |
| intersect(LocSet, LocSet) | |
| allFields(Object) | set of all locations for the given object |
| allObjects(Field) | set of all locations for the given field; e.g., $\{(d, f) \mid \text{f.a. } d \in D^{\texttt{Object}}\}$ |
| arrayRange(Object, int, int) | set representing all array locations in the specified range (both inclusive) |

# Translating Assignable Clauses—Example

**Example**

```
assignable \everything;
```

is translated into the DL term

# Translating Assignable Clauses—Example

**Example**

```
assignable \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

# Translating Assignable Clauses—Example

**Example**

```
assignable \everything;
```

is translated into the DL term

$$\texttt{allLocs}$$

**Example**

```
assignable this.next, this.content[5..9];
```

is translated into the DL term

$$\begin{array}{l}\texttt{union}(\texttt{singleton}(\texttt{self}, \texttt{next}), \\ \qquad \texttt{arrayRange}(\texttt{self.content}, 5, 9))\end{array}$$

# Translating JML into Intermediate Format

**Intermediate format for contract of method** *m*

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

with

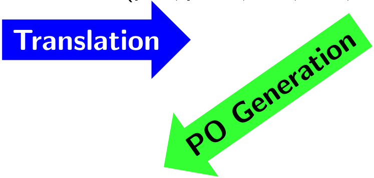
- a precondition DL formula *pre* ✔,
- a postcondition DL formula *post* ✔,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$ ✔,
- a variant *var* a term of type `any` (postponed),
- a modifies set *mod*, either of type `LocSet` or `\strictly_nothing` ✔

# From JML Contracts via Intermediate Format to Proof Obligations (PO)

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

Intermediate Format

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



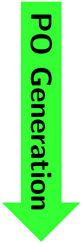

Proof obligation as DL formula

$$pre \rightarrow \langle \texttt{this.m(params);} \rangle (post \wedge frame)$$

# Generating a PO from the Intermediate Format: Idea

Given intermediate format of contract of `m` implemented in class `C`:

$$(\mathit{pre}, \mathit{post}, \mathtt{TOTAL}, \mathit{var}, \mathit{mod})$$

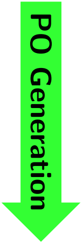



$$\mathit{pre} \rightarrow \langle \mathtt{self.m(args)} \rangle (\mathit{post} \ \wedge \ \underbrace{\mathit{frame}}_{\text{correctness of assignable}} \ )$$

# Generating a PO from the Intermediate Format: Idea

Given intermediate format of contract of `m` implemented in class `C`:

$$(pre, post, \mathtt{TOTAL}, var, mod)$$




$$pre \rightarrow \langle \mathtt{self.m(args)} \rangle (post \ \land \ \underbrace{frame}_{\substack{\text{correctness of} \\ \text{assignable}}} \ )$$

In case of $div = \mathtt{PARTIAL}$, box modality is used

# Generating a PO from Intermediate Format: Method Identification

$$pre \rightarrow \langle \texttt{self.m(args)} \rangle (post \ \wedge \ frame)$$

# Generating a PO from Intermediate Format: Method Identification

$$pre \rightarrow \langle \texttt{self.m(args)} \rangle (post \ \wedge \ frame)$$

- Dynamic dispatch: `self.m(...)` causes split into all possible implementations

# Generating a PO from Intermediate Format: Method Identification

$$pre \rightarrow \langle \texttt{self.m(args)} \rangle (post \ \land \ frame)$$

- Dynamic dispatch: `self.m(...)` causes split into all possible implementations
- Special statement Method Body Statement:

  $\texttt{m(}\mathit{args}\texttt{)@C}$

  Meaning: implementation of `m` in class `C`

# Generating a PO from Intermediate Format: Exceptions

$$\mathit{pre} \rightarrow \langle \texttt{self.m(args)@C} \rangle (\mathit{post} \ \land \ \mathit{frame})$$

Postcondition *post* states either

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

but: $\langle \textbf{throw} \ \texttt{exc;} \rangle \varphi$ is trivially false

# Generating a PO from Intermediate Format: Exceptions

$$\mathit{pre} \rightarrow \langle \texttt{self.m(args)@C} \rangle (\mathit{post} \ \wedge \ \mathit{frame})$$

Postcondition *post* states either

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

but: $\langle \textbf{throw} \ \texttt{exc;} \rangle \varphi$ is trivially false

How to refer to an exception in post-state?

# Generating a PO from Intermediate Format: Exceptions

$$pre \rightarrow \langle \texttt{self.m(args)@C} \rangle (post \ \wedge \ frame)$$

Postcondition *post* states either

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

but: $\langle \textbf{throw} \ \texttt{exc;} \rangle \varphi$ is trivially false

How to refer to an exception in post-state?

$$pre \rightarrow \left\langle \begin{array}{l} \texttt{exc = } \textbf{null}\texttt{;} \\ \textbf{try} \ \texttt{\{} \\ self.m(args)@C \\ \texttt{\}} \ \textbf{catch} \ \texttt{(Throwable e)\{exc = e;\}} \end{array} \right\rangle (post \ \wedge \ frame)$$

Recall: generation of *post* (pp.22,23) uses program variable `exc`

# The Generic Precondition *genPre*

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

is still not complete.

# The Generic Precondition *genPre*

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

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
- ...

# The Generic Precondition *genPre*

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

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
- ...

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

# The Generic Precondition *genPre*

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

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
- ...

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

Idea: Formalise assumption as additional precondition *genPre*

$$(genPre \wedge pre) \rightarrow \\ \langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{self.m(args)@C\}}\ \mathbf{catch}\ \ldots\ \rangle(post\ \wedge\ frame)$$

# The Generic Precondition *genPre* (background info)

$$
\begin{aligned}
\textit{genPre} := \quad & \texttt{wellFormed(heap)} \\
& \wedge \texttt{self} \neq \textbf{null} \\
& \wedge \texttt{self.} <\!\texttt{created}\!> = \texttt{TRUE} \\
& \wedge \texttt{C} :: \texttt{exactInstance(self)} \\
& \wedge \textit{paramsInRange}
\end{aligned}
$$

- `wellFormed(h)`: predefined predicate;
  true iff `h` is regular Java heap
- `C :: exactInstance(o)`: predefined predicate;
  true iff `o` has exact type `C` (not just subtype of `C`)
- *paramsInRange* formula stating that method arguments are in range

# The Generic Precondition *genPre*

$(genPre \wedge pre) \rightarrow$
$\quad \langle$`exc=`**null**`;` **try** `{self.m(args)@C}` **catch** $\ldots\ \rangle(post\ \wedge\ frame)$

is still not complete.

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

# The Generic Precondition *genPre*

$$(\mathit{genPre} \land \mathit{pre}) \rightarrow \\ \langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{self.m(args)@C\}}\ \mathbf{catch}\ \ldots\ \rangle(\mathit{post}\ \land\ \mathit{frame})$$

is still not complete.

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

$$(\mathit{genPre} \land \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\} \\ \langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{self.m(args)@C\}}\ \mathbf{catch}\ \ldots\ \rangle(\mathit{post}\ \land\ \mathit{frame})$$

Recall: `heapAtPre` was used in translation of `\old`, p.21

# Generating a PO from Intermediate Format: The *frame* DL Formula

$$(\mathit{genPre} \land \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\textbf{null}\texttt{;}\ \textbf{try}\ \texttt{\{self.m(args)\}}\ \textbf{catch}\ \ldots\ \rangle$$
$$(\mathit{post}\ \land\ \mathit{frame})$$

If $\mathit{mod} =$ `\strictly_nothing` then *frame* is defined as:

$$\forall o; \forall f; (o.f = o.f@\texttt{heapAtPre})$$

# Generating a PO from Intermediate Format: The *frame* DL Formula

$$(\mathit{genPre} \land \mathit{pre}) \rightarrow \{\mathtt{heapAtPre} := \mathtt{heap}\}$$
$$\langle \mathtt{exc}\texttt{=}\mathbf{null};\ \mathbf{try}\ \{\mathtt{self.m(args)}\}\ \mathbf{catch}\ \ldots\ \rangle$$
$$(\mathit{post}\ \land\ \mathit{frame})$$

If *mod* is a location set, then *frame* is defined as:

$$\forall o; \forall f; (\quad (o, f) \in \{\mathtt{heap} := \mathtt{heapAtPre}\}\mathit{mod} \\ \lor\ o.\texttt{<created>@heaptAtPre} = \mathtt{FALSE} \\ \lor\ o.f = o.f\mathtt{@heapAtPre} \quad )$$

# Generating a PO from Intermediate Format: The *frame* DL Formula

$$(\mathit{genPre} \wedge \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\textbf{null}\texttt{;}\ \textbf{try}\ \texttt{\{self.m(args)\}}\ \textbf{catch}\ \ldots\ \rangle$$
$$(\mathit{post}\ \wedge\ \mathit{frame})$$

If *mod* is a location set, then *frame* is defined as:

$$\forall o; \forall f; (\ (o, f) \in \{\texttt{heap} := \texttt{heapAtPre}\}\mathit{mod} \\ \vee\ o.\texttt{<created>@heaptAtPre} = \texttt{FALSE} \\ \vee\ o.f = o.f\texttt{@heapAtPre}\ )$$

Says that every location $(o, f)$ either

- belongs to the modifies set (evaluated in the pre-state), or
- was not (yet) created before the method invocation, or
- holds the same value before and after the method execution

# Generating a PO from Intermediate Format: Result Value

$$(\mathit{genPre} \land \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{self.m(args)\}}\ \mathbf{catch}\ \ldots\ \rangle$$
$$(\mathit{post}\ \land\ \mathit{frame})$$

is still not complete.

- For non-void methods, need to refer to result in *post*

# Generating a PO from Intermediate Format: Result Value

$$(\mathit{genPre} \wedge \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{self.m(args)\}}\ \mathbf{catch}\ \ldots\ \rangle (\mathit{post} \wedge \mathit{frame})$$

is still not complete.

- For non-void methods, need to refer to result in *post*

$$(\mathit{genPre} \wedge \mathit{pre}) \rightarrow \{\texttt{heapAtPre} := \texttt{heap}\}$$
$$\langle \texttt{exc=}\mathbf{null}\texttt{;}\ \mathbf{try}\ \texttt{\{result = self.m(args)\}}\ \mathbf{catch}\ \ldots\ \rangle (\mathit{post} \wedge \mathit{frame})$$

Recall: `\result` was translated to program variable `result`, see p.20

# Examples

Demo