Software Engineering using Formal Methods

Proof Obligations

Wolfgang Ahrendt, Mauricio Chimento

14 October 2016

This Part

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,
- and proving

DL proof obligations from JML specifications

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) {..}
}
```

Translation

Intermediate Format

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

PO Generation

Proof obligation as DL formula

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

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

Tho 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 `\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;`
   - `exceptional_behavior` the clause `ensures false;`

Normalisation: Making Implicit Specifications Explicit

Implicit Specifications

- Kind of behavior
- `non_null` by default
- Implicit `\invariant_for(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(Object o)` add `requires o != 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 `\invariant_for(this)` as `requires`, `ensures` & `signals` clause

Making `\invariant_for(this)` explicit for method specifications

1. Add explicit `\invariant_for(this)` to non-helper method specs, as
   - `requires \invariant_for(this);`
   - `ensures \invariant_for(this);`
   - `signals (Throwable t) \invariant_for(this);`
2. Deactivate implicit `\invariant_for(this)` by adding `helper` modifier to method (if not already present)

Normalisation: Example

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

Normalisation: Example

- Kind of behavior

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

Normalisation: Example

- `non_null` by default

```
/*@ 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

```
/*@ 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

```
/*@ 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: 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.

Normalisation:Clause Contraction

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 *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.

`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

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

`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  &  select(heap,x,<created>)=TRUE  &  F(e_0))
   -> F(e_1))

F((\exists T x; e_0; e_1)) =
 \exists T x; (
      (!x=null  &  select(heap,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

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

- `\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 $\texttt{Object}::\texttt{<inv>}(\texttt{heap}, o)$ 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* ✔?almost,
- 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`

Translation of Ensures Clauses

What is missing for `ensures` clauses?

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

Translation of Ensures Clauses

Translating `\result`

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

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

where `result` $\in$ *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. 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

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

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 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 Signals and Ensures to *post*

The DL formula *post* is then defined as

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

Note:

Normalisation of `normal_behavior` contract gives

`signals (Throwable exc) false;`

Then *post* is:

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

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

Translating JML into Intermediate Format

The Divergence Indicator

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

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 *var* a term of type `any` (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`

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

Definition (Locations)

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

Note: Location is a semantic and not a syntactic entity.

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

$$\mathtt{allLocs}$$

Example

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

is translated into the DL term

$$\mathtt{union}(\mathtt{singleton}(\mathtt{self}, \mathtt{next}),\\ \mathtt{arrayRange}(\mathtt{select}(\mathtt{heap}, \mathtt{self}, \mathtt{context}), 5, 9))$$

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 `any` (postponed),
- a modifies set *mod*, either of type `LocSet` or `\strictly_nothing` ✔

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) {..}
}
```

Translation

Intermediate Format

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

PO Generation

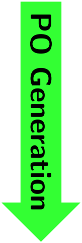
Proof obligation as DL formula

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

Generating a PO from the Intermediate Format: Idea

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

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



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

(In case of $\mathit{div} = \texttt{PARTIAL}$, box modality is used)

Generating a PO from Intermediate Format: Method Identification

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

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

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

Meaning: Placeholder for the method body of class `C`

Generating a PO from Intermediate Format: Exceptions

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

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?

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

(Recall: Normalistion and *post*-generation used program variable `exc`)

The Generic Precondition *genPre*

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

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*

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

The Generic Precondition *genPre*

$$
\begin{aligned}
\textit{genPre} := \quad & \texttt{wellFormed}(\texttt{heap}) \\
& \land \texttt{paramsInRange} \\
& \land \texttt{self} \neq \textbf{null} \\
& \land \texttt{boolean} :: \texttt{select}(\texttt{heap}, \texttt{self}, \texttt{<created>}) = \texttt{TRUE} \\
& \land \texttt{C} :: \texttt{exactInstance}(\texttt{self}) \\
& \land \texttt{exc} = \textbf{null}
\end{aligned}
$$

- `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*

$(genPre \land pre) \to$
$\langle$`exc=`**null**`;` **try** `{self.m(args)@C}` **catch** ... $\rangle$(*post* & *frame*)

is still not complete.

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

$(genPre \land pre) \to \{\texttt{heapAtPre} := \texttt{heap}\}$
$\langle$`exc=`**null**`;` **try** `{self.m(args)@C}` **catch** ... $\rangle$(*post* & *frame*)

(Reminder: `heapAtPre` was used in translation of `\old` in *post*)

Generating a PO from Intermediate Format: Result Value

$(genPre \land pre) \rightarrow \{\mathtt{heapAtPre} := \mathtt{heap}\}$
$\langle$`exc=`**null**`;` **try** `{self.m(args)}` **catch** ... $\rangle$
$(post \quad \& \quad frame)$

is still not complete.

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

$(genPre \land pre) \rightarrow \{\mathtt{heapAtPre} := \mathtt{heap}\}$
$\langle$`exc=`**null**`;` **try** `{result = self.m(args)}` **catch** ... $\rangle$
$(post \quad \& \quad frame)$

(Reminder: `result` was used in translation of `\result` in *post*)

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

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

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

$$\forall o; \forall f; (\mathtt{select}(\mathtt{heapAtPre}, o, f) = \mathtt{select}(\mathtt{heap}, o, f))$$

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

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

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

$$\begin{aligned} \forall o; \forall f; (\quad & \mathtt{select}(\mathtt{heaptAtPre}, o, \mathtt{<created>}) = \mathtt{FALSE} \\ & \lor \mathtt{select}(\mathtt{heapAtPre}, o, f) = \mathtt{select}(\mathtt{heap}, o, f) \\ & \lor (o, f) \in \{\mathtt{heap} := \mathtt{heapAtPre}\}\mathit{mod}) \end{aligned}$$

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'