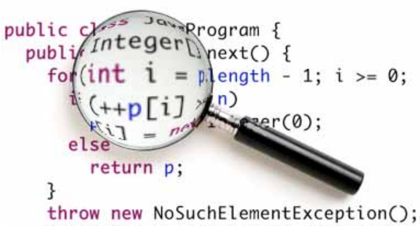
15-819M: Data, Code, Decisions

10: Proof Obligations

André Platzer

aplatzer@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, PA



```
public class JavaProgram {
  public Integer[] next() {
    for(int i = p.length - 1; i >= 0;
      if (++p[i] > n)
        p[i] = new Integer(0);
      else
        return p;
    }
    throw new NoSuchElementException();
```

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. Generating Proof Obligations
4. Translating JML to DL
5. Schematic POs
6. Literature

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. Generating Proof Obligations
4. Translating JML to DL
5. Schematic POs
6. Literature

Connections and Putting Pieces Together

making the connection between

JML

and

Dynamic Logic / KeY

Connections and Putting Pieces Together

making the connection between

JML

and

Dynamic Logic / KeY

- generating,

Connections and Putting Pieces Together

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,

Connections and Putting Pieces Together

making the connection between

JML

and

Dynamic Logic / KeY

- generating,
- understanding,
- and proving

DL proof obligations from JML specifications

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. Generating Proof Obligations
4. Translating JML to DL
5. Schematic POs
6. Literature

Tutorial Example

We follow 'KeY Quicktour for JML'   (cited below as [KQJ])

Sources: KeY Quicktour on course page

Scenario: simple `PayCard`

Inspecting JML Specification

inspect `quicktour/jml/paycard/PayCard.java`

[KQJ, 2.2]

JML Feature: Nested Specification Cases

method `charge()` has nested specification case:

```
@ 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;
@ |}
```

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;
```

JML Feature: `assignable \nothing`

method `charge()` has exceptional behavior case:

```
@ public exceptional_behavior
@ requires amount <= 0;
@ assignable \nothing;
```

JML Feature: `assignable \nothing`

method `charge()` has exceptional behavior case:

```
@ public exceptional_behavior
@ requires amount <= 0;
@ assignable \nothing;
```

`assignable \nothing` prohibits side effects

Difference to **`pure`**:

- **`pure`** also prohibits non-termination
- **`assignable`** clause is local to specification case (here: local to **`exceptional_behavior`**)

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. **Generating Proof Obligations**
4. Translating JML to DL
5. Schematic POs
6. Literature

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**

  assuming less invariants:
  - is fully sound
  - can compromise provability

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**

  assuming less invariants:
  - is fully sound
  - can compromise provability

  sometimes invariants of *other* classes also needed (select class+inv.)

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**

  assuming less invariants:
    - is fully sound
    - can compromise provability

  sometimes invariants of *other* classes also needed (select class+inv.)
- select contract which **modifies** `balance`

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**

  assuming less invariants:
  - is fully sound
  - can compromise provability

  sometimes invariants of *other* classes also needed (select class+inv.)
- select contract which **modifies** `balance`

  (in JML: **`modifies`** synonymous for **`assignable`**)

Generating Proof Obligations (POs)

generate **EnsuresPost** PO for normal behavior of `charge()`

[KQJ, 3.1+3.2]

- start KeY prover
- in `quicktour/jml`, open `paycard`
- select `PayCard.charge` and **EnsuresPost**
- inspect **Assumed Invariants**
  assuming less invariants:
    - is fully sound
    - can compromise provability

  sometimes invariants of *other* classes also needed (select class+inv.)
- select contract which **modifies** `balance`
  (in JML: **`modifies`** synonymous for **`assignable`**)
- **Current Goal** pane displays proof obligation as DL sequent

Generating Proof Obligations

For loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

generate **EnsuresPost** PO for normal behavior of `isValid()`

Generating Proof Obligations

For loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

generate **EnsuresPost** PO for normal behavior of `isValid()`

generate **EnsuresPost** PO for exceptional behavior of `charge()`

Generating Proof Obligations

For loading more proof obligations:
re-open **Proof Obligation Browser** under **Tools** menu

generate **EnsuresPost** PO for normal behavior of `isValid()`

generate **EnsuresPost** PO for exceptional behavior of `charge()`

generate **PreservesOwnInv** PO for `charge()`

expressing that `charge()` preserves all invariants (of its own class)

[KQJ, 4.3.1+4.3.2]

Outline

Translating JML to DL POs

principles of translating JML to proof obligations in DL

- issues in translating arithmetic expressions
- translating `this`
- identifying the method's implementation
- translating boolean JML expressions to first-order logic formulas
- translating preconditions
- translating class invariants
- translating postconditions
- storing `\old` fields prior to method invocation
- storing actual parameters prior to method invocation
- expressing that 'exceptions are (not) thrown'
- *putting everything together*

Translating JML to DL POs

DISCLAIMER:

The following presentation is

- incomplete
- not fully precise
- over-simplistic
- omitting details/complications
- deviating from exact implementation in KeY

Translating JML to DL POs

DISCLAIMER:

The following presentation is

- incomplete
- not fully precise
- over-simplistic
- omitting details/complications
- deviating from exact implementation in KeY

enable you to read/understand proof obligations

Translating JML to DL POs

DISCLAIMER:

The following presentation is

- incomplete
- not fully precise
- over-simplistic
- omitting details/complications
- deviating from exact implementation in KeY

enable you to read/understand proof obligations

Issues on Translating Arithmetic Expressions

Often:

- KeY replaces arithmetic JAVA operators by generalized operators, generic towards various integer semantics (JAVA, Overflow-checks, Mathematics),
  example: "`+`" becomes "`javaAddInt`"
- KeY inserts casts like `(jint)`,
  needed for type hierarchy among primitive types,
  example: "`0`" becomes "`(jint)(0)`"

Issues on Translating Arithmetic Expressions

Often:

- KeY replaces arithmetic JAVA operators by generalized operators, generic towards various integer semantics (JAVA, Overflow-checks, Mathematics),
  example: "`+`" becomes "`javaAddInt`"
- KeY inserts casts like `(jint)`,
  needed for type hierarchy among primitive types,
  example: "`0`" becomes "`(jint)(0)`"

(no need to memorize this)

Translating `this`

- explicit, and
- implicit

`this` reference translated to explicit `self` reference

Translating `this`

- explicit, and
- implicit

`this` reference translated to explicit `self` reference

e.g., given class

```
public class MyClass {
  ...
  private int f;
  ...
}
```

Translating `this`

- explicit, and
- implicit

`this` reference translated to explicit `self` reference

e.g., given class

```
public class MyClass {
  ...
  private int f;
  ...
}
```

- `f` translated to `self.f`
- `this.f` translated to `self.f`

Identifying Method Implementations

JAVA's dynamic dispatch selects a method's implementation *at runtime*

Identifying Method Implementations

JAVA's dynamic dispatch selects a method's implementation *at runtime*

For a method call `m(args)`,
KeY models selection of implementation from `package.Class` by
`m(args)@package.Class`

Identifying Method Implementations

JAVA's dynamic dispatch selects a method's implementation *at runtime*

For a method call `m(`*args*`)`,
KeY models selection of implementation from `package.Class` by
`m(`*args*`)@package.Class`

Example

`charge(x)@paycard.PayCard`
executes class `paycard.PayCard`'s implementation of method call
`charge(x)`

Translating Boolean JML Expressions

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

JML

- Formulas are no separate syntactic category
- Instead:
  JAVA's `boolean` expressions extended with first-order concepts (quantifiers)

KeY logic

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

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

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

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

`b_0`, `b_1` `boolean` JML expressions

`e_0`, `e_1` JAVA expressions

$\mathcal{T}$ may add '`self.`' or '`@ClassName`' (see pp.21,22)

$\mathcal{E}$ may add casts, transform operators (see p.20)

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

$\mathcal{F}$((\forall T x; e_0)) = \forall T x;
!x=null -> $\mathcal{F}$(e_0)

$\mathcal{F}$((\exists T x; e_0)) = \exists T x;
!x=null & $\mathcal{F}$(e_0)

$\mathcal{F}$((\forall T x; e_0; e_1)) = \forall T x;
!x=null & $\mathcal{F}$(e_0)
-> $\mathcal{F}$(e_1)

$\mathcal{F}$((\exists T x; e_0; e_1)) = \exists T x;
!x=null & $\mathcal{F}$(e_0) & $\mathcal{F}$(e_1)

Translating Preconditions

If selected contract *Contr* has preconditions

```
@ requires b_1;
@ ...
@ requires b_n;
```

they are translated to

Translating Preconditions

If selected contract *Contr* has preconditions

```
@ requires b_1;
@ ...
@ requires b_n;
```

they are translated to

$$\mathcal{PRE}(\mathit{Contr}) \\ = \\ \mathcal{F}(\texttt{b\_1})\ \texttt{\&}\ \texttt{...}\ \ \texttt{\&}\ \mathcal{F}(\texttt{b\_n})$$

Translating Class Invariants

Invariant

```
class C {
   ...
   //@ invariant inv_i;
   ...
}
```

is translated to

Translating Class Invariants

Invariant

```
class C {
  ...
  //@ invariant inv_i;
  ...
}
```

is translated to

$$\mathcal{INV}(\texttt{inv\_i})$$

$$=$$

$$\texttt{\textbackslash forall C o; ((o.<created> = TRUE \& !o = null) ->}$$
$$\{\texttt{self:=o}\}\mathcal{F}(\texttt{inv\_i}))$$

Translating Postconditions

If selected contract *Contr* has postconditions

```
@ ensures b_1;
@ ...
@ ensures b_n;
```

they are translated to

Translating Postconditions

If selected contract *Contr* has postconditions

```
@ ensures b_1;
@ ...
@ ensures b_n;
```

they are translated to

$$\mathcal{POST}(\mathit{Contr}) \\ = \\ \mathcal{F}(\texttt{b\_1})\ \texttt{\&}\ \texttt{...}\ \ \texttt{\&}\ \mathcal{F}(\texttt{b\_n})$$

Translating Postconditions

If selected contract *Contr* has postconditions

```
@ ensures b_1;
@ ...
@ ensures b_n;
```

they are translated to

$$\mathcal{POST}(Contr) = \mathcal{F}(\texttt{b\_1})\ \texttt{\&}\ \texttt{...}\ \ \texttt{\&}\ \mathcal{F}(\texttt{b\_n})$$

Special treatment of expressions in post-condition: next slide

Translating Expressions in Postconditions

Assume assignable clause

```
@ assignable <assignable_fields>;
```

Translating Expressions in Postconditions

Assume assignable clause

`@ assignable <assignable_fields>;`

translating expressions in postconditions (interesting cases only):

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

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

$\mathcal{E}_{old}$ defined like $\mathcal{E}$, with the exception of:

$$\begin{aligned} \mathcal{E}_{old}(\texttt{e.f}) &= \texttt{fAtPre}(\mathcal{E}_{old}(\texttt{e})) \\ \mathcal{E}_{old}(\texttt{f}) &= \texttt{fAtPre(self)} \end{aligned}$$

for f $\in$ *<assignable_fields>*

Translating Expressions in Postconditions

Assume assignable clause

`@ assignable <assignable_fields>;`

translating expressions in postconditions (interesting cases only):

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

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

$\mathcal{E}_{old}$ defined like $\mathcal{E}$, with the exception of:

$$\begin{aligned} \mathcal{E}_{old}(\texttt{e.f}) &= \texttt{fAtPre}(\mathcal{E}_{old}(\texttt{e})) \\ \mathcal{E}_{old}(\texttt{f}) &= \texttt{fAtPre(self)} \end{aligned}$$

for $\texttt{f} \in$ *<assignable_fields>*

'`fAtPre`' meant to refer to field '`f`' *in the pre-state*

Storing Pre-State of a Field

For an **`assignable`** field `f` of class `C`

```
class C {
  ...
  private T f;
  ...
}
```

translation of postcondition replaces `f` in `\old(..)` by `fAtPre` (p.29)
TODO: store pre-state values of `f` in `fAtPre`

Storing Pre-State of a Field

For an **`assignable`** field `f` of class `C`

```
class C {
  ...
  private T f;
  ...
}
```

translation of postcondition replaces `f` in **`\old`**`(..)` by `fAtPre` (p.29)
TODO: store pre-state values of `f` in `fAtPre`

$$\mathcal{STORE}(\mathtt{f})$$
$$=$$

Storing Pre-State of a Field

For an **`assignable`** field `f` of class `C`

```
class C {
  ...
  private T f;
  ...
}
```

translation of postcondition replaces `f` in **`\old`**`(..)` by `fAtPre` (p.29)
TODO: store pre-state values of `f` in `fAtPre`

$$\mathcal{STORE}(\mathtt{f})$$

$$=$$

`\for C o; fAtPre(o) := o.f`

Storing Pre-State of a Field

For an **`assignable`** field `f` of class `C`

```
class C {
  ...
  private T f;
  ...
}
```

translation of postcondition replaces `f` in **`\old`**`(..)` by `fAtPre` (p.29)
TODO: store pre-state values of `f` in `fAtPre`

$$\mathcal{STORE}(\mathtt{f})$$

$$=$$

`\for C o; fAtPre(o) := o.f`

Not a formula, but

Storing Pre-State of a Field

For an **assignable** field f of class C

```
class C {
  ...
  private T f;
  ...
}
```

translation of postcondition replaces f in **\old**(..) by fAtPre (p.29)
TODO: store pre-state values of f in fAtPre

$$\mathcal{STORE}(\texttt{f})$$

$$=$$

`\for C o; fAtPre(o) := o.f`

Not a formula, but a quantified update

Storing Pre-State of All Assignable Fields

If selected contract *Contr* has preconditions

```
@ assignable f_1, ..., f_n;
```

then pre-state of *all* assignable fields can be stored by

Storing Pre-State of All Assignable Fields

If selected contract *Contr* has preconditions

```
@ assignable f_1, ..., f_n;
```

then pre-state of *all* assignable fields can be stored by *one* parallel update:

Storing Pre-State of All Assignable Fields

If selected contract *Contr* has preconditions

`@ assignable f_1, ..., f_n;`

then pre-state of *all* assignable fields can be stored by *one* parallel update:

$$\mathcal{STORE}(\mathit{Contr}) \\ = \\ \{\ \mathcal{STORE}(\texttt{f\_1})\ \texttt{||}\ \ldots\ \texttt{||}\ \mathcal{STORE}(\texttt{f\_n})\ \}$$

Expressing Normal Termination

How to express in DL:
method call `m()` will not throw an exception

Expressing Normal Termination

How to express in DL:
method call `m()` will not throw an exception
(if method body from class `C` in package `p` is invoked)

Expressing Normal Termination

How to express in DL:
method call `m()` will not throw an exception
(if method body from class `C` in package `p` is invoked)

```
\<{ exc = null;
    try {
      m()@p.C;
    }  catch (java.lang.Throwable e) {
      exc = e;
    }
  }\> exc = null
```

Expressing Normal Termination

How to express in DL:
method call `m()` will not throw an exception
(if method body from class `C` in package `p` is invoked)

```
\<{ exc = null;
    try {
      m()@p.C;
    }  catch (java.lang.Throwable e) {
      exc = e;
    }
  }\> exc = null
```

Note difference:

- Java assignments
- equation, i.e., formula (in KeY output format)

Expressing Exceptional Termination

How to express in DL:
method call `m()` will throw an exception

Expressing Exceptional Termination

How to express in DL:
method call `m()` will throw an exception
(if method body from class `C` in package `p` is invoked)

Expressing Exceptional Termination

How to express in DL:
method call `m()` will throw an exception
(if method body from class `C` in package `p` is invoked)

```
\<{ exc = null;
    try {
      m()@p.C;
    }  catch (java.lang.Throwable e) {
      exc = e;
    }
  }\> !exc = null
```

Expressing Exceptional Termination

How to express in DL:
method call `m()` will throw an exception
(if method body from class `C` in package `p` is invoked)

```
\<{ exc = null;
    try {
      m()@p.C;
    }  catch (java.lang.Throwable e) {
      exc = e;
    }
  }\> !exc = null  & <typing of exc>
```

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. Generating Proof Obligations
4. Translating JML to DL
5. **Schematic POs**
6. Literature

PO for Normal Behavior Contract

PO for a normal behavior contract *Contr* for `void` method `m()`, with chosen assumed invariants `inv_1`, ..., `inv_n`

```
==>
     INV(inv_1)
   & ...
   & INV(inv_n)
   & PRE(Contr)
 -> STORE(Contr)
     \<{ exc = null;
         try {
           m()@p.C;
         }  catch (java.lang.Throwable e) {
           exc = e;
         }
       }\> exc = null & POST(Contr)
```

PO for Normal Behavior Allowing Non-Termination

PO for a normal behavior contract *Contr* for method `m()`,
where *Contr* has clause `diverges true;`

```
==>
     𝓘𝓝𝓥(inv_1)
   & ...
   & 𝓘𝓝𝓥(inv_n)
   & 𝓟𝓡𝓔(Contr)
 -> 𝓢𝓣𝓞𝓡𝓔(Contr)
     \[{ exc = null;
         try {
           m()@p.C;
         }  catch (java.lang.Throwable e) {
           exc = e;
         }
       }\] exc = null & 𝓟𝓞𝓢𝓣(Contr)
```

PO for Normal Behavior of Non-Void Method

PO for a normal behavior contract *Contr* for `non-void` method `m()`,

```
==>
      𝓘𝓝𝓥(inv_1)
    & ...
    & 𝓘𝓝𝓥(inv_n)
    & 𝓟𝓡𝓔(Contr)
 -> 𝓢𝓣𝓞𝓡𝓔(Contr)
      \<{ exc = null;
          try {
            result = m()@p.C;
          }  catch (java.lang.Throwable e) {
            exc = e;
          }
        }\> exc = null & 𝓟𝓞𝓢𝓣(Contr)
```

PO for Normal Behavior of Non-Void Method

PO for a normal behavior contract *Contr* for `non-void` method `m()`,

```
==>
      $\mathcal{INV}$(inv_1)
    & ...
    & $\mathcal{INV}$(inv_n)
    & $\mathcal{PRE}$(Contr)
 -> $\mathcal{STORE}$(Contr)
      \<{ exc = null;
          try {
            result = m()@p.C;
          }  catch (java.lang.Throwable e) {
            exc = e;
          }
        }\> exc = null & $\mathcal{POST}$(Contr)
```

recall: $\mathcal{POST}(Contr)$ translated **`\result`** to `result` (p.29)

PO for Preserving Invariants

assume method `m()` has contracts $Contr_1$, …, $Contr_j$

PO stating that:

Invariants `inv_1`, ..., `inv_n` are preserved
in all cases covered by a contract.

```
==>
     𝓘𝓝𝓥(inv_1) & ... & 𝓘𝓝𝓥(inv_n)
   & ( 𝓟𝓡𝓔(Contr_1) | ...  | 𝓟𝓡𝓔(Contr_1) )
 -> \[{ exc = null;
       try {
         m()@p.C;
       }  catch (java.lang.Throwable e) {
         exc = e;
       }
     }\] 𝓘𝓝𝓥(inv_1) & ... & 𝓘𝓝𝓥(inv_n)
```

Examples

Follow the quicktour with KeY and understand examples

Outline

1. Proof Obligations
2. Tutorial Example: PayCard
3. Generating Proof Obligations
4. Translating JML to DL
5. Schematic POs
6. Literature

Literature for this Lecture

Essential

KeY Quicktour see course page