Formal Methods for Java

Lecture 17: Advanced Key

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

December 21, 2011

Abnormal Termination in Java

Abnormal termination in Java is caused by

- a `break` statement,
- a `continue` statement,
- a `return` statement,
- a `throw` statement, or
- a statement that throws a exception.

Reasoning about exceptions

How can we express that statement α throws an exception?

- The trick is to put an exception handler into the code:

$$\langle \{ \text{Throwable } \mathit{thrown} = \mathit{null}; \\ \quad \textbf{try } \{\alpha;\} \\ \quad \textbf{catch } (\text{Throwable } \mathit{ex}) \{ \mathit{thrown} = \mathit{ex}; \} \} \rangle \mathit{thrown} \neq \mathit{null}$$

Reasoning with try-catch blocks

When an exception is thrown, the surrounding try blocks become important:

```
\find( \<{ .. try { throw #se; #slist1 }
             catch (#t #v0) { #slist2 } ... }\> post )
```

1. throwing a handled exception: `#se instanceof #t`
   ```
   \replacewith( \<{ .. #t #v0 = #se; #slist2 ... }\> post )
   ```
2. throwing an unhandled exception: `! (#se instanceof #t)`
   ```
   \replacewith( \<{ .. throw #se; ... }\> post )
   ```
3. throwing a null pointer: `#se = null`
   ```
   \replacewith( \< { .. try { throw new NullPointerExc(); #slist1
                        catch (#t #v0) { #slist2 } ... }\> post )
   ```

The KeY system defines a single rule:

```
\replacewith( \< { .. if (#se = null) then
                        try { throw new NullPointerExc(); #slist1
                        catch (#t #v0) { #slist2 }
                      else if (#se instanceof #t) then
                        #t v0 = #se; #slist2
                      else throw #se;
                    ... }\> post )
```

Throw without try-catch blocks

If the surrounding block is not a try block, the block is just removed:

```
\find( \<{ .. #label: { throw #se; #slist1 } ... }\> post )
\replacewith( \<{ .. throw #se; ... }\> post )
```

If there is no surrounding block it depends on modality:

1. total correctness:
   ```
   \find( \<{ throw #se }\> post )
   \replacewith( false )
   ```

2. partial correctness:
   ```
   \find( \[{ throw #se }\] post )
   \replacewith( true )
   ```

Runtime exceptions

Instructions that throw exceptions are converted to a `throw` instruction:

```
\find( \<{ .. #v[#se]=#se0 ... }\> post )
```

- Normal Execution `#v != null`

```
\add( !#v = null &
      #se < #v.length & #se >= 0 &
      arrayStoreValid(#v, #se0) ==>)
\replacewith( \{#v[#se] := #se0\}\<{ .. ...}\> post )
```

- Null Reference `#v == null`

```
\add( #v = null ==> )
\replacewith( \<{ .. throw new NullPointerException(); ...}\> post )
```

- Index Out Of Bounds:

```
\add( !#v = null &
      #se >= #v.length | #se < 0 ==>)
\replacewith( \<{ .. throw new ArrIdxOOBException(); ...}\> post )
```

- Array Store Exception:

```
\add( !#v = null &
      #se < #v.length & #se >= 0 &
      !arrayStoreValid(#v, #se0) ==>)
\replacewith( \<{ .. throw new ArrayStoreException(); ...}\> post )
```

Abnormal termination by `break`

The handling of `break` statements is very similar to `try-catch`:

- If the surrounding block has that label, the break is executed:
  ```
  \find( \<{ .. #label: { break #label; #slist1 } ... }\> post )
  \replacewith( \<{ .. ... }\> post )
  ```
- If the surrounding block has not the right label the block is removed.
  ```
  \find( \<{ .. #label2: { break #label; #slist1 } ... }\> post )
  \replacewith( \<{ .. break #label; ... }\> post )
  ```
- The same for `try-catch` blocks:
  ```
  \find( \<{ .. try { break #label; #slist1 }
                catch (#t #v) { #slist2 } ... }\> post )
  \replacewith( \<{ .. break #label; ... }\> post )
  ```

Loops with `break`/`continue`

`break`/`continue` statements are translated to labelled `break`.

rule loop_unwind:

```
\find( \<{ .. while (#expr) {.... continue; .... break; ....} ... }\> post
\replacewith( \<{ .. if (#expr) {
                       #lab1: {
                         #lab2: {
                            ....
                            continue #lab2;
                            ....
                            break #lab1;
                            ....
                         }
                         while (#expr) {.... continue; .... break; ....}
                     } ... } \> post)
```

KeY and Procedures

In KeY, the default rule is to inline the procedures.
Advantages:

- No function contract needed.
- No separate proof for correctness of function needed.

But it has several disadvantages:

- Proof gets larger (especially important if proof is interactive).
- Proof has to be repeated for every function call.
- No recursive procedures possible.

The rule Use Operation Contract

The rule "Use Operation Contract" allows compositional proofs.
It opens three subgoals:

- Pre: Show that pre-condition holds (this includes class invariants).
- Post: Show that with the post-condition, the remaining program is correct.
- Exceptional Post: Show that if called method throws an exception, the remaining program is correct.

Note: Use Operation Contract cannot be used for the method you are just proving.