Formal Methods for Java

Lecture 12: Object Invariants

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

November 30, 2012

The Invariant Problem

```
public class SomeClass {
  /*@ invariant inv; @*/

  /*@ requires P;
    @ ensures Q;
    @*/
  public void doSomething() {

    assume(P);
    assume(inv);

    ...code of doSomething...

    assert(Q);
    assert(inv);

  }
}
```

```
public class OtherClass {
  public void caller(SomeObject o) {
    ...some other code...

    assert(P);

    o.doSomething();

    assume(Q);

  }
}
```

- ESC/Java checks the highlighted assumes and asserts.
- This is unsound!

Why Unsound?

The following rule is unsound:

$$\frac{\{P \land \mathit{inv}\}\ \mathit{doSomething}()\ \{Q \land \mathit{inv}\}}{\{P\}\ \mathit{doSomething}()\ \{Q\}}$$

This is also not the intuition...

What is the Intuition?

An invariant should hold (almost) always.

$$\frac{\{\mathit{true}\}\ \texttt{some other code}\ \{P\}}{\{\mathit{true} \land \mathit{inv}\}\ \texttt{some other code}\ \{P \land \mathit{inv}\}}$$

- Only sound, if `some other code` cannot change truth of invariant.
- For example, invariant depends only on private fields

Invariants Depend on Other Objects

Consider a doubly linked list:

```
class Node {
  Node prev, next;
  /*@ invariant this.prev.next == this && this.next.prev == this; @*/
}
class List {
  public void add() {
    Node newnode = new Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
  }
}
```

The invariant of `this` depends on the fields of `this.next` and `this.prev`.
Moreover the `List.add` function changes the fields of the invariants of `Node`.

The List example

First observation: The invariant should be put into the List class:

```
class Node {
  Node prev, next;
}
class List {
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
                n.prev.next == n && n.next.prev == n); @*/
  public void add() {
    Node newnode = new Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

The List example

Second observation: Node objects must not be shared between two different lists.

```
class Node {
  /*@ ghost Object owner; @*/
  Node prev, next;
}
class List {
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
                 n.prev.next == n && n.next.prev == n
                 && n.owner == this); @*/
  public void add() {
    Node newnode = new Node();
    //@ set newnode.owner = this;
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

The List example

Third observation: One may only change the owned fields.

```
class Node {
  /*@ ghost Object owner; @*/
  Node prev, next;
}
class List {
  Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
                 n.prev.next == n && n.next.prev == n
                 && n.owner == this); @*/
  public void add() {
    Node newnode = new Node();
    //@ set newnode.owner = this;
    newnode.prev = first.prev;
    newnode.next = first;
    //@ assert(first.prev.owner == this)
    first.prev.next = newnode;
    //@ assert(first.owner == this)
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

The Owner-as-Modifier Property

JML supports the owner-as-modifier property, when invoked as `jmlc --universes`. The underlying type system is called Universes.

- The class *Object* has a ghost field *owner*.
- Fields can be declared as `rep`, `peer`, `readonly`.
  - `rep` *Object x* adds an implicit invariant (or requires) *x.owner* = `this`.
  - `peer` *Object x* adds an implicit invariant (or requires) *x.owner* = `this`.*owner*.
  - `readonly` *Object x* do not restrict owner, but do not allow modifications.
- The `new` operation supports `rep` and `peer`:
  - `new /*@rep@*/Node()` sets owner field of new node to `this`.
  - `new /*@peer@*/Node()` sets owner field of new node to `this`.*owner*.

The List with Universes Type System

```
class Node {
  /*@ peer @*/ Node prev, next;
}
class List {
  /*@ rep @*/ Node first;
  /*@ private ghost JMLObjectSet nodes; @*/
  /*@ invariant (\forall Node n; nodes.has(n);
                 n.prev.next == n && n.next.prev == n
                 && n.owner == this); @*/
  public void add() {
    Node newnode = new /*@ rep @*/ Node();
    newnode.prev = first.prev;
    newnode.next = first;
    first.prev.next = newnode;
    first.prev = newnode;
    //@ set nodes = nodes.insert(newnode);
  }
}
```

The Universes Type System

A simple type system can check most of the ownership issues:

- `rep` *T* can be assigned without cast to `rep` *T* and `readonly` *T*.
- `peer` *T* can be assigned without cast to `peer` *T* and `readonly` *T*.
- `readonly` *T* can be assigned without cast to `readonly` *T*.

One need to distinguish between the type of a field `peer Node prev` and the type of a field expression: `rep Node first.prev`.

- If `obj` is a `peer` type and `fld` is a `peer` *T* field
  then `obj.fld` has type `peer` *T*.
- If `obj` is a `rep` type and `fld` is a `peer` *T* field
  then `obj.fld` has type `rep` *T*.
- If `obj = this` and `fld` is a `rep` *T* field
  then `this.fld` has type `rep` *T*.
- In all other cases `obj.fld` has type `readonly` *T*.

`readonly` References

To prevent changing readonly references there are these restrictions:
If *obj* has type `readonly` *T* then

- *obj.fld* = *expr* is illegal.
- *obj.method*(...) is only allowed if *method* is a pure method.

It is allowed to cast `readonly` *T* references to `rep` *T* or `peer` *T*:

- (`rep` *T*) *expr* asserts that *expr.owner* == `this`.
- (`peer` *T*) *expr* asserts that *expr.owner* == `this`.*owner*.

Modification only by Owner

All write accesses to a field of an object are

- in a function of the owner of the object or
- in a function of a object having the same owner as the object that was invoked (directly or indirectly) by the owner of the object.

An invariant that only depends on fields of owned objects can only be invalidated by the owner or the function it invokes.