Formal Methods for Java

Lecture 15: Object Invariants

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

June 21, 2017

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:

$$\frac{\{P \wedge inv\}\ doSomething()\ \{Q \wedge inv\}}{\{P\}\ doSomething()\ \{Q\}}$$

This is also not the intuition...

What is the Intuition?

An invariant should hold (almost) always.

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

- 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.

Otherwise, *obj.fld* = *expr* is legal iff *expr* can be cast to the type of *obj.fld*.

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.

The Invariant Problem

There are some problems with invariants:

- Ownership: invariants can depend on fields of other objects.
  For example, the invariant of list accesses node fields.
- Callback: invariants can be temporarily violated.
  While invariant is violated we call a different method that calls back.
- Atomicity: invariants can be temporarily violated.
  While invariant is violated another thread accesses object.

Temporarily Violating Invariants

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/

  public void add(int v) {
    /* 1 */
    size++;
    /* 2 */
    if (size > content.length) {
      newContent = new int[2*size+1];
      ...
      content = newContent;
    }
    ...
    /* 3 */
  }
}
```

When do Invariants Hold?

- Before a public method is called. `/* 1 */`
- After a public method returns. `/* 3 */`
- However, it may be violated in between. `/* 2 */`

Private Methods

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/

  private void growContent() {

  private /*@ helper @*/ void growContent() {

    ...
    content = newContent;
  }

  public void add(int v) {
    /* invariant should hold */
    size++;
    /* invariant may be violated */
    if (size > content.length)
      growContent();
    ...
    /* invariant should hold, again */
  }
}
```

- Sometimes an invariant should not hold for a private method.
- JML has the keyword `/*@ helper @*/`.

Calling Methods of Other Classes

```
public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/

  public void add(int v) {
    /* invariant should hold */
    size++;
    /* invariant may be violated */
    if (size > content.length) {
      newContent = new int[2*size+1];
      System.arraycopy(content, 0, newContent, 0, content.length);
      content = newContent;
    }

    ...
    /* invariant should hold, again */
  }
}
```

- The invariant need not to hold, when calling other methods.
- However there is the callback problem.

The Callback Problem

```
public class Log {
  public void log(String p) {
    logfile.write("Log: "+p+" list is "+Global.theList);
} }

public class Container {
  int[] content;
  int size;
  /*@ invariant 0 <= size && size <= content.length; @*/

  public void add(int v) {
    /* invariant should hold */
    size++;
    /* invariant may be violated */
    if (size > content.length) {
      Logger.log("growing array.");
    ...
  }

  public String toString() {
    /* invariant should hold */
    ...
} }
```

The Callback Problem

- A method of a different class can be called while invariant is violated.
- This method may call a method of the first class.
- Who has to ensure that the invariant holds?
- jmlrac complains that invariant does not hold
- ESC/Java checks that most invariants hold at every method call, but not all invariants; this may lead to unsoundness.

A Ghost Variable for Invariants

Idea of David A. Naumann and Mike Barnett:

- Make the places where an invariant does not hold explicit.
- Add a ghost variable *packed* that indicates if the invariant should hold.
- Before modifying an object set this variable to `false`.
- When modification is finished, set it to `true`.
- The following invariant should always hold:
  packed ==> invariants of object
- The caller has to ensure that the objects he uses are packed.

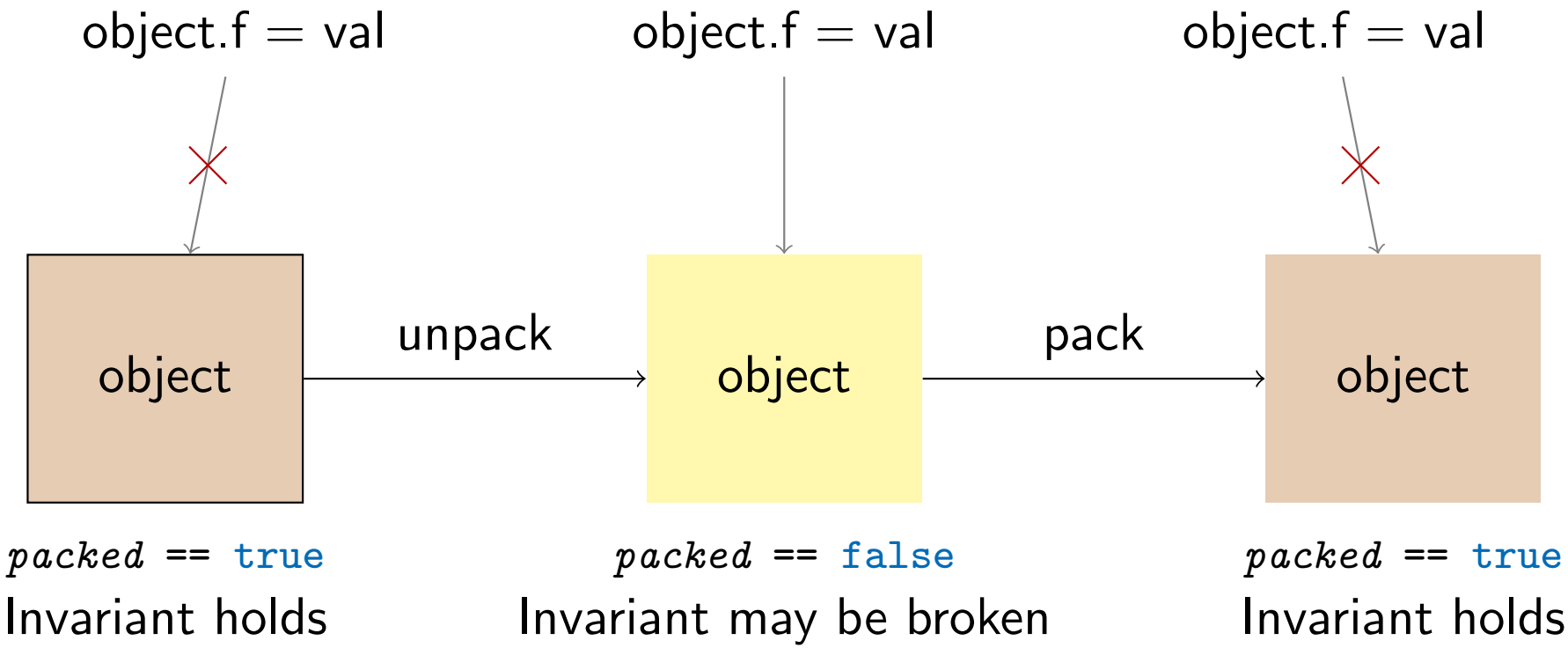
Example: A Ghost Variable for Invariants

```
//@ public ghost boolean packed;
//@ private invariant packed ==> (size >= 0 && size <= content.length);

/*@ requires packed;
  @ ensures packed;
  @*/
public void add(int v) {
  unpack this;
  size++;
  ...
  pack this;
}
```

- The pre- and post-conditions explicitly states that invariant holds
- `unpack this` is an abbrevation for:
  ```
  assert this.packed;
  set this.packed = false;
  ```
- `pack this` is an abbrevation for:
  ```
  assert !this.packed;
  assert /*invariant of this holds*/;
  set this.packed = true;
  ```

The pack/unpack Mechanism



- An object must be unpacked before fields may be accessed.
- The invariant has to hold only while object is packed.
- The invariant may only depend on fields of the object.

Checking with Atomicity

Static Checking with *packed* ghost field:

- Fields may only be modified if *packed* is false.
- For each `pack` operation check that invariant holds again.
- Thus *packed* `==>` *invariants* holds for all states.

Tree Example

```
class TreeNode {
  int key, value;
  TreeNode left, right;
  /*@ invariant left != null ==> left.key <= key; @*/
  /*@ invariant right != null ==> right.key >= key; @*/

  public void add(Node n) {
    if (n.key < key) {
      if (left == null)
        left = n;
      else
        left.add(n);
    } else {
      ...
    }
  }
}
```

Adding Packed variable

```
class TreeNode {
  int key, value;
  TreeNode left, right;
  //@ public ghost boolean packed = false;

  /*@ invariant packed ==> (left != null ==> left.key <= key); @*/
  /*@ invariant packed ==> (right != null ==> right.key >= key); @*/

  //@ requires packed;
  //@ ensures packed;
  public void add(/*@non_null@*/ TreeNode n) {
    // unpack this
    if (n.key < key) {
      if (left == null)
        left = n;
      else
        left.add(n);
    } else {
      ...
    }
    // pack this
  }
}
```

Running ESC/Java gives:

```
> escjava2 -q TreeNode.java
TreeNode.java:19: Warning: Precondition possibly not established (Pre)
                left.add(n);
                         ^
Associated declaration is "TreeNode.java", line 9, col 8:
    //@ requires packed;
```

The nodes *left* and *right* must be packed!

Fixing the invariant

```
class TreeNode {
  int key, value;
  TreeNode left, right;
  //@ public ghost boolean packed = false;

  /*@ invariant packed ==> (left != null ==>
                            left.packed && left.key <= key); @*/
  /*@ invariant packed ==> (right != null ==>
                            right.packed && right.key >= key); @*/

  //@ requires packed;
  //@ ensures packed;
  public void add(/*@non_null@*/ TreeNode n) {
    ...
  }
}
```

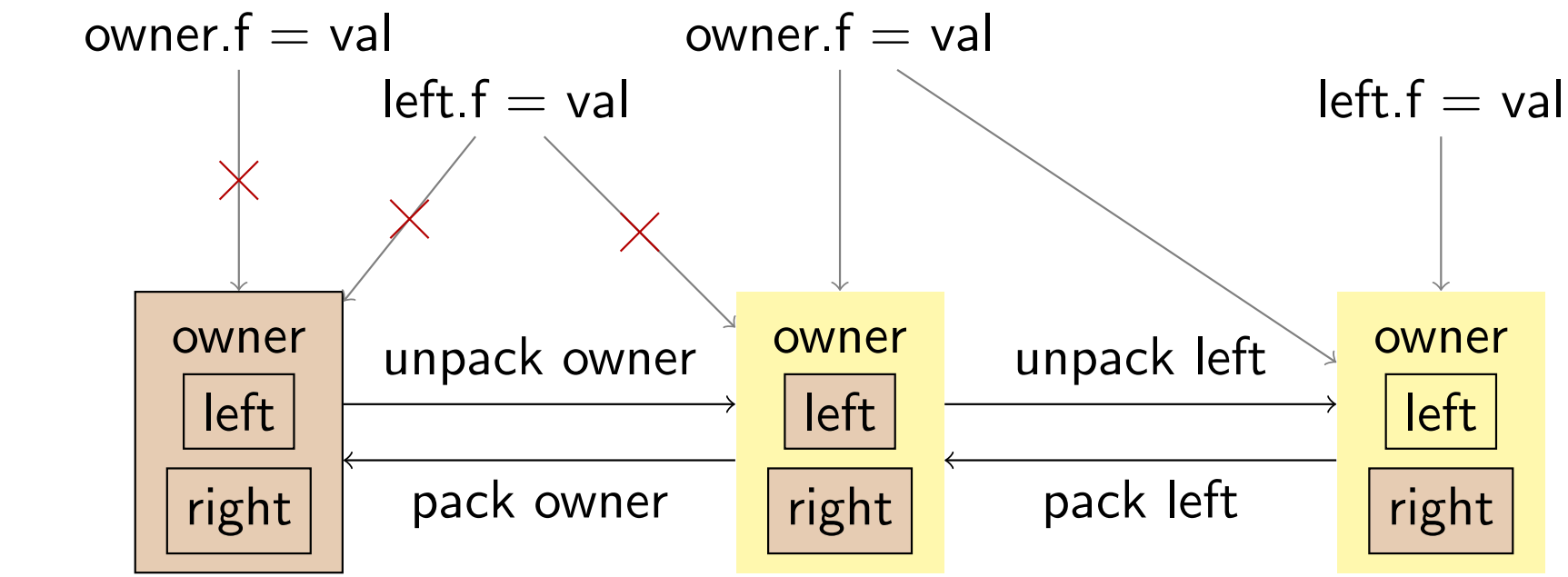
Adding Ownership

There are still problems:

- The invariant also depends on fields of *left* and *right*.
  In particular the *left.key* and *left.packed*.
- Can `unpack this` violate the invariant of another TreeNode?
- How can we exclude undesired sharing,
  e.g., *left* == `this` or *left* == *n*?

Solution: Use the ownership principle

Ownership and pack/unpack



- The owner must be unpacked before an owned object can be unpacked.
- The invariant of owner may depend on owned objects.

Ownership And pack/unpack

How does pack/unpack work with ownership?

- To modify an object, you must `unpack` it first.
- To `unpack` an object, you must `unpack` the owner.
- To `pack` the owner again, its invariant must hold.

`unpack` *obj* is an abbreviation for:

```
assert(obj.packed);
assert(obj.owner == null || !obj.owner.packed);
set obj.packed = false;
```

`pack` *obj* ensures that its owned classes are packed.

```
assert(!obj.packed);
assert(left != null ==> (left.owner == this && left.packed));
assert(right != null ==> (right.owner == this && right.packed));
assert(/* other invariants of obj holds*/);
set obj.packed = true;
```

Adding Ownership

```
class TreeNode {
  int key, value;
  TreeNode left, right;
  //@ public ghost Object owner;
  //@ public ghost boolean packed = false;

  /*@ invariant packed ==> (left != null ==>
          left.owner == this && left.packed && left.key <= key); @*/
  /*@ invariant packed ==> (right != null ==>
          right.owner == this && right.packed && right.key >= key); @*/

  /*@ requires packed && (owner == null || !owner.packed) &&
    @          n.packed && n.owner == null;
    @ ensures packed; */
  public void add(/*@non_null@*/ TreeNode n) {
    ...
  }
}
```