

Formal Methods for Java

Lecture 13: Invariants with Pack and Unpack

Jochen Hoenicke



Software Engineering
Albert-Ludwigs-University Freiburg

December 4, 2012

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

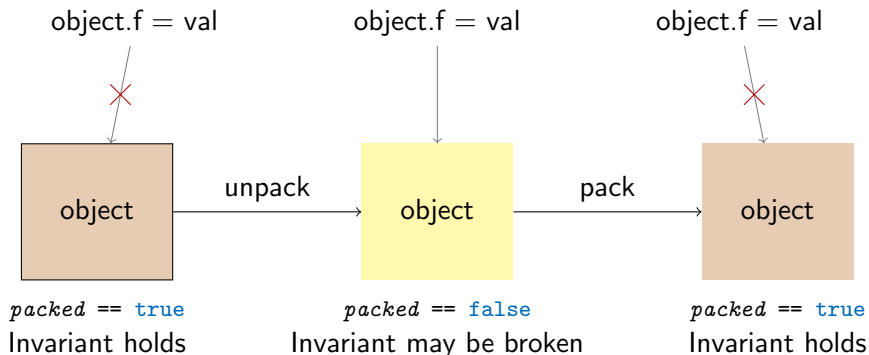
```
//@ public ghost boolean packed;  
//@ private invariant packed ==> (size >= 0 && size <= content.length);  
  
/*@ requires packed;  
  @ ensures packed;  
  @*/  
public void add(int v) {  
    unpack this;  
    size++;  
    ...  
    pack this;  
}
```

- The pre- and post-conditions explicitly states that invariant holds
- `unpack this` is an abbreviation for:

```
assert this.packed;  
set this.packed = false;
```
- `pack this` is an abbreviation 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.

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 $packed \implies invariants$ 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) {
    ...
  }
}
```

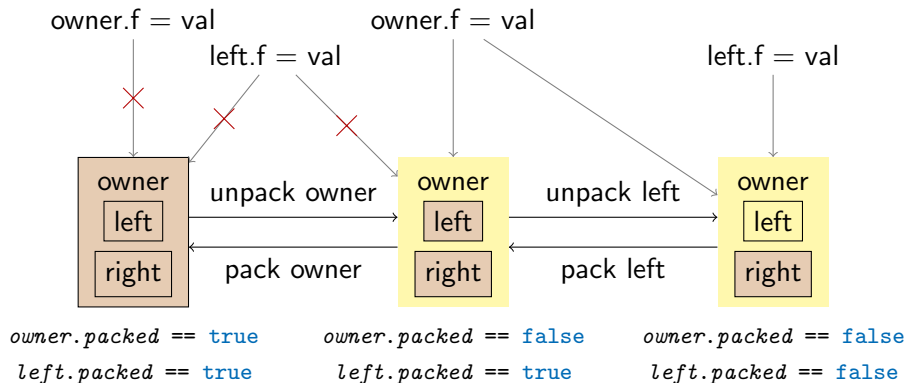
Adding Ownership

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

Ownership vs. Friendship

The ownership discipline has a few restrictions.

- An object invariant can only depend on fields of owned objects.
- An object can have at most one owner.
- A field may only be changed by the owner, or if the owner is unpacked.

Sometimes too restrictive!

Friendship offers another way to depend on other objects:

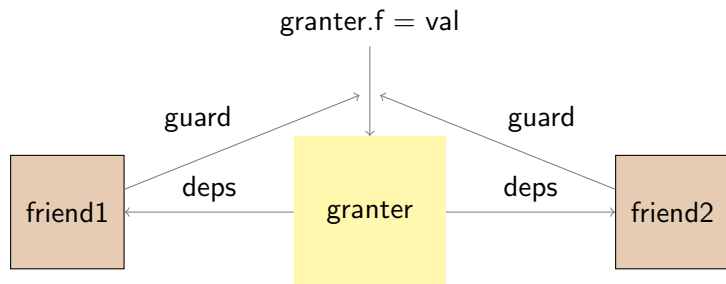
- An invariant can also depend on fields of granters.
- The class must define update guards for all fields it depends on.
- A granter object has a list of friends that depend on fields.
- A field may be changed if the update guards of all friends holds.

Friendship is not symmetric. The allies are:

- Granter G that gives rights to depend on a field.
- Friend C whose invariant depends on a field.

Every class that changes a field of G has to check the friend's update condition.

Friendship and field accesses



- Friend's invariant can depend on granted fields.
- Access to granted fields is checked against update guards.
- A granter can have many friends.
- All current friends must be checked.
- The friend objects can be packed or unpacked.

Friendship Example

```
class Object {
  /*@ spec_public @*/ int hashCode;
  /*@ friend Map reads hashCode;
  /*@ ghost JMLObjectSet deps;
}

class Map {
  JMLObjectSet buckets[];
  /*@ invariant
  \forall int i ; 0 <= i && i < buckets.length;
  (\forall Object o; buckets[i].has(o); o.deps.has(this) &&
  Math.abs(o.hashCode % buckets.length) == i); @*/

  /*@ guard obj.hashCode := val by
  val % buckets.length == obj.hashCode % buckets.length; @*/
}
```

Update Guard and Invariant

```
class FriendClass {  
    //@ invariant friendInvariant(granter.field)  
    //@ guard granter.field := val by updateGuardForField(granter, val);  
}
```

The update guard must guarantee that the invariant is not invalidated:
 $friends.packed \ \&\& \ friendInvariant(granter.field)$
 $\ \&\& \ updateGuardForField(granter, val) \ ==> \ friendInvariant(val)$

What May Appear in an Invariant

Only the following field accesses are allowed in an invariant:

- `this.field` for all fields.
- `x.field` if it appears in a subformula:
`\forall Object x ; x.owner == this ==> ...`
- `object.field` if `object != null && object.owner == this` can be proven.
- `x.field` if it appears in a subformula:
`\forall Object x ; x.deps.has(this) ==> ...`
- `object.field` if `object != null && object.deps.has(this)` can be proven.

Why Is This Sound?

A field access $obj.f=val$ only affects invariants of

- obj ,
- $obj.owner$ if it is not null,
- and the objects in $obj.deps$.

obj and $obj.owner$ must be unpacked if field is accessed. Thus their invariants need not to hold afterwards.

For the objects in $obj.deps$ the update guard must hold. Therefore, the invariant holds also with the new value val for $obj.f$.