/*@ immutable @*/ objects

Erik Poll

SoS group (aka the LOOP group)
Radboud University Nijmegen

KeY workshop – June 2005

yet another JML keyword...

- Java provides final – ie. immutable - fields
  What about immutable objects?

- It would be nice to have a notion of immutable object, that
  - can be specified in JML,
  - statically enforced,
  - guarantees immutability, and
  - can be exploited in program verification...

overview

- Why would we want immutable objects ?
- How do we enforce immutability ?
- How to exploit immutability ?

why immutability ?

- Good software engineering practice

  "immutable objects greatly simplify your life"

  Knowing that an object is immutable rules out
  - problems with aliasing
  - problems with race conditions
- Performance
  - compiler optimisations, no need for synchronisation
- Specification
  - immutability is an important integrity property
  - eg. immutability of Strings, URLs, Permissions, etc. vital for security

why immutability ?

Useful in program verification

```
char[] a;
String s;
....
if (s.equals("abc")) {
      a[1]='d';
      //@ assert s.equals("abc");
     }
...
```

Knowing that strings are immutable allows us to prove this.

why immutability ?

JML has a library of – supposedly immutable - model classes, for mathematical objects such as sets, relations,

```
//@ public model JMLObjectSet s;

//@ requires ! s.contains(o);
//@ ensures  s.equals(\old(s).union(o));
public void addListener(Object o) { ... }
```

Enforcing immutability

starting point: `pure`

JML has notion of pure to specify absence of side-effects:

- pure method has no side-effects
- pure constructor has no side-effects, except on newly allocated state
- pure class only has pure methods, pure constructors, and pure sub-classes

pure does not imply immutable

```
public /*@ pure @*/ class Integer{
  public int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

methods of an Integer object don't have any side-effects, but maybe methods of some other class have side-effects an Integer object's state

is this pure class immutable ?

```
public /*@ immutable?? @*/ class Integer {
  private int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

Still not immutable, because field i is not final:
an object created with `new Integer(5)` may be observed to change from 0 to 5 in a multi-threaded program

counterexample

Thread 1 creates object

```
x = new Integer(5);
```

This takes three steps:

5. a new Integer object is allocated, with i field 0
6. i field is set to 5
7. x is set to point to this newly allocated object

Steps 2 and 3 can be reordered by compiler or VM!

Thread 2 observes this object

```
int j = x.getValue();
```

Thread 2 may observe value 0, namely if it observes x after 3 and before 2.

fields must be final to ensure immutability

```
public /*@ immutable @*/ class Integer {
  private final int i;
  public Integer(int j){ i = j; }
  public int getValue(){ return i; }
}
```

This class has immutable objects, thanks to the newly revised Java Memory Model (JSR-133, 2004)

People tend to forget final declarations...

final fields may still be mutable...

```
public /*@ immutable?? @*/ class Integer {
  public static Integer latest;
  private final int i;
  public Integer(int j){ i = j;
                          latest = this;} // leaks
  public int getValue(){ return i;)
}
```

Constructor leaks `this`, hence field `i` not immutable:
`Integer(5)` may be observed to change from 0 to 5.

There are a few more ways to leak `this`

ensuring immutability

A pure class is immutable if

1. all instance fields are final, and
2. constructors don't leak `this`

This definition is implicit in JSR-133, but it is not strong enough if we want immutable objects with immutable sub-objects

what about sub-objects?

```
public /*@ immutable @*/ BankTransfer {
  final Integer amount;
  final byte[] transferID;
  final BankAccount src, dest;
 ...
}
```

- `amount` and `transferID` objects part of the `Banktransfer` object
- `src` and `dest` objects probably not

what about sub-objects?

```
public /*@ immutable @*/ BankTransfer {
  final Integer amount;
  final byte[] transferID;
  final BankAccount src, dest;
 ...
}
```

- `amount` and `transferID` objects part of the `Banktransfer` object, and should also be immutable
- `src` and `dest` objects probably not, and may be mutable

specifying the "state" of an object

```
public /*@ immutable @*/ BankTransfer {
  final /*@ rep @*/ Integer amount;
  final /*@ rep @*/ byte[] transferID;
  final BankAccount src, dest;
 ...
}
```

Sub-object `amount` and `tranferID` should be immutable.

- This means `transferID` should not be aliased!
- JML universes type system – or some other form of alias control/confinement/ownership – guarantees this.

- `amount` can be aliased, because it's immutable.

ensuring immutability

A pure class is immutable if

1. all instance fields are final, and
2. constructors don't leak `this`, and
3. all instance fields that are references either
   i. have immutable types, or
   ii. are part of the "state" and cannot be aliased, or
   iii. are excluded from the "state"

still not enough...

```
public /*@ immutable?? @*/ StrangeInteger {
  private final int i;
  StrangeInteger(int j){ i = j; }
  int getValue(){ return SomeClass.someStaticField;}
}
```

As well as specifying and checking
what a method writes (assignable aka modifies clauses)
we also need to check
what a method reads (readable clauses)

Ensuring immutability

Def. A pure class is immutable if

1. all instance fields are final, and
2. constructors don't leak `this`, and
3. all instance fields that are references
   i. have immutable types, or
   ii. are part of the state and cannot be aliased, or
   iii. are excluded from the "state"
4. its methods don't read mutable state (outside its own state)

Related work on enforcing immutability

- Javari [Birka & Ernst at MIT, OOPSLA'04]
  - proposal to add readonly modifier to Java
  - more refined notion of immutability, eg allowing both mutable and immutable (readonly) references to the same object
  - doesn't deal with sub-objects (3) or reading mutable state(4)

- Jan Schäfer at TU Kaiserslautern
  - system for enforcing immutability
  - forgets check on leaking this (2)

Exploiting immutability

exploiting immutability

Immutability is easily to exploit in

- alias control system
- relaxing synchronisation in multi-threaded programs

How about exploiting immutability in verification ?

observational immutability

- Example: `bankTransfer.getAmount()` is a constant
- object is "observationally immutable" if we cannot observe any mutation by invoking its methods
- if o is observationally immutable, then

  o.m(x1,...,xn)

  always returns the same result, if xi are primitive values or immutable objects

exploiting immutability in verification?

A method

```
C m(C1 x1, ..., Cn xn)
```

is interpreted/modeled as function

```
m : GlobalState×Ref×C1×..×Cn –> C
```

For immutable objects we can omit state argument

```
m :              Ref×C1×..×Cn –> C
```

if all Ci are primitive or immutable types

Implemented by David Cok in ESC/Java2

exploiting immutability in verification?

```
public /*@ immutable @*/ class Integer {
  ...
  public Integer add(Integer i) {
    return new Integer(getValue()+i.getValue);
  }
```

Here we get `add: Ref x Ref -> Ref`

But this means

$$i == j \Rightarrow k.add(i) == k.add(j)$$

not

$$i.equals(j) \Rightarrow k.add(i).equals(k.add(j))$$

which is what we'd really want...

exploiting immutability in verification?

- Trick to exploit immutability by ommiting state argument is perfect if arguments and result have primitive types

- But if result is a reference type, it may not be sound.
  Eg `add` always returns the same result, but here the same means the same modulo == , not `.equals`

- If an argument is of reference type, it is not complete
  Eg `add` always returns the same result for .equal arguments, not just == arguments

alternative approaches

- We could specify the properties of an immutable type as axioms to the back-end theorem prover.
- We could also give a native implementation of the immutable Integer class in our back-end theorem prover.
- But how do we know this is sound ?

maybe we also want /*@constant@*/ methods?

(Mutable) object can have "constant" methods which always return the same result

For example

```
public class Object {
...
public /*@ constant @*/ int hashCode(){...};
public /*@ constant @*/ Class getClass(){...};
...
}
```

conclusions & future work

- Immutability is nice property, that deserves to be documented, if not in Java then in JML:

  stresses design decision; specifies important integrity property; enables checks that people don't forget `final`; simplifies alias control & synchronisation.
- Enforcing immutability is possible, but complicated
  - requires alias control and readable clauses (in addition to assignable/modifies clauses)
- Exploiting immutability in verification is tricky, except for primitive types
  - Can we devise a provably sound approach ?