Different ways of annotating and proving a quicksort with Jack

Julien Charles

Thursday, february the 16th 2006

Introduction

- Goal : verify a quicksort written in Java annotated with JML with Jack
- Static verification of this program :
  $\longrightarrow$ done in an interactive way with Coq.
- Motivation : need for some simple examples in Jack
  - Already done in ESC/Java2
  - Already done in Krakatoa
    $\longrightarrow$ Now doing it in Jack

Jack

The Java Modelling Language (JML) is used to annotate the Java programs we want to verify with Jack.

- Uses a weakest precondition calculus to generate proof obligations
- The proof obligations are decomposed w.r.t. the different execution cases
- Fully integrated in Eclipse as a plugin, and use a plugin architecture to include new provers
- To prove the quicksort we used the Coq Output from Jack
- Coq plugin in can be both used to prove automatically and interactively the proof obligations generated by Jack

Coq with Jack

The files generated by Jack in Coq can be separated into 3 categories :

- The prelude containing the logic used :
  - A static part (jack.arith.v jack.references.v jack.tactics.v)
  - A dynamic part generated for a specified class (myClass.classes.v myClass.subtypes.v myClass.v)
- The proof obligation (1 file)
- A file containing some custom tactics written by the user

To edit the files we use the CoqEditor plugin, an integrated editor for Coq files in Eclipse.

General Plan

1. Annotating in a normal way
2. Using purity
3. Using 'native' types

Annotating in a normal way

1. The QuickSort algorithm
2. Annotations
3. Kind of proof obligations

The QuickSort algorithm...

```
private void sort(int lo, int hi) {
    int left, right, pivot;
    if( !(lo < hi)) return;
    left = lo;
    right = hi;
    pivot = tab[hi];
    while(left < right) {
        while((left < right) && (tab[left] <= pivot)) left++;
        while((left < right) && (tab[right] >= pivot)) right--;
        if(left < right){
          swap(left, right);
        }
    }
```

...in Java

```
swap(left, hi) ;
if (left > 0)
   sort(lo, left -1) ;
if (left +1 < tab.length)
   sort(left +1, hi) ;
}
```

- Property wanted : it sorts the array
- The permutation property is left out, we use just an injection property

Annotations (1)

Specifications of the main method :

```
/*@ requires (tab != null) && (0 <= lo) && (lo < tab.length) &&
  @       (0 <= hi) && (hi < tab.length);
  @ modifies tab[lo .. hi];
  @ ensures (\forall int i, j; (lo <= i) && (i <= hi) ==> (lo <= j) && (j <= hi)
  @       ==> (i < j) ==> tab[i] <= tab[j]) &&
  @       (\forall int i; lo <= i && i <= hi; (\exists int j; lo <= j && j <= hi &&
  @                   \old(tab[j]) == tab[i]));
  @*/
  private void sort(int lo, int hi)
```

Annotations (2)

Annotations of the loop invariant :

```
/*@ loop_modifies left, right, tab[lo..(hi - 1)] ;
  @ loop_invariant (lo <= left) && (left <= right) && (right <= hi) &&
  @       (\forall int m ; (lo <= m) && (m < left) ==> tab[m] <= pivot)
  @       && (\forall int n ; (right < n) && (n <= hi) ==> pivot <= tab[n])
  @       && tab[right] >= pivot &&
  @       (\forall int i ; lo <= i && i <= hi - 1 ;
  @                 (\exists int j ; lo <= j && j <= hi && \old(tab[j]) == tab[i])) ;
  @ decreases (right - left) ;
  @*/
```

Kind of proof obligations

- Around 230 proof obligations (POs)
- 100 are solved automatically
- Difficulties :
  - the loop termination wasn't so trivial
    $\longrightarrow$ the use of ghost variables was necessary
  - the proof can easily get messy
    $\longrightarrow$ use of assertions
  - the verbosity ot the annotations

Using purity

1. Purity in JML
2. Purity in Jack
3. Simplifyng pure methods
4. Specification macros, predicates for JML
5. A quicksort annotated with predicates

Purity in JML

Pure methods are methods you can use in your specification in JML.

- No 'visible' side effect
- A constructor can be pure

$\longrightarrow$ We'd like to keep track of the pure method name in the proof obligations

Purity in Jack

- In Jack constructors are not pure
- At first pure methods were directly unfolded within the proof obligations
- Now there is a couple definition / hypothesis

```
Definition myfun_1 : A1Type → A2Type → ResType → Prop :=
fun (h arg1 arg2 Res) => Context1 → Res = true. (* ResType = bool *)

Definition myfun_2 : A1Type → A2Type → ResType → Prop :=
fun (h arg1 arg2 Res) => Context2 → Res = false.
...
(* Used within a hypothesis : *)
... → myfun_2 this l_var1 Res_23 → Res_23 = true.
```

Specification macros, predicates in JML

- Pure methods can be used to do some specification macros :

```
/*@ requires true;
  @ ensures \result == ((tab != null) && (i >= 0) && (i < tab.length));
  @*/
public/*@ pure @*/ boolean withinBounds(Object[] tab, int i) {
      return (tab != null) && (i >= 0) && (i < tab.length);
}
```

  We'd like to prove lemmas on this method to ease the proofs :
  $\longrightarrow$ directly define pure methods in Coq

Coq predicates in JML

How the predicates are defined for Jack :

- Within JML :

  ```
  //@ public native boolean withinBounds(Object[] tab, int i);
  ```

- In the file userTactics.v :

  ```
  Definition withinBounds :
      (Reference → t_int → Reference) → Reference → t_int → bool :=
      fun intelements tab value =>
          andb (notb (REFeq tab null))
          (andb (Zle_bool 0 value) (Zlt_bool value (arraylength tab))).
  ```

A quicksort annotated with predicates

All the important annotations are replaced by these predicates :

```
/*@ public native boolean is_sorted(int[] tab, int beg, int end) ;
  @ public native boolean is_inf(int [] tab, int value, int beg, int end) ;
  @ public native boolean is_sup(int [] tab, int value, int beg, int end) ;
  @ public native boolean withinBounds(int[] tab, int i) ;
  @*/
```

These predicates are indeed not so useful by themselves :

- They permit to have clearer annotations withinBounds(tab, i) instead of (i >= 0) && (i < tab.length)
- We can prove properties on them at the Coq level

→ Some properties are still hard to express (permutation on arrays)

Using native types

1. Native types
2. Model variables
3. Quicksort with model

Native types

- To express the permutation property :
  $\longrightarrow$ Have list in Coq, not in JML (since there is a list library in Coq...)
- A new construct to directly use a Coq defined datatype :
  //@ public native class IntList
  In the file userTactics.v :
  Definition IntList : list t_int.
- The types are not some standard Java/JML class types :
  - it does not inherit from the class object
  - it is not an instance, etc...
  - more of a functional type : modifiers create new objects and are 'static'

A list library

We can now define a list library binded on some Coq types, to use in annotations :

```
/*@ public native class IntList {
  @       public static native IntList create() ;
  @       public static native IntList cons(int i, IntList c) ;
  @       public static native IntList app(IntList c1, IntList c2) ;
  @       public native boolean is_in(int i) ;
  @       public native boolean equals(IntList i) ;
  @       public native int length() ;
  @ }
  @*/
```

→ Link it with the program

Model variables

- Model variables are specification variables associated with a program variable.
- The association is done through a translation function.
- In JML it has the following syntax :
  - //@ model MyType modVar ;
  - //@ represents modVar $\leftarrow$ P(progVar)
- You cannot do a 'set' with these variables, i.e. you cannot assign them.
  $\rightarrow$ The model variable is considered to be modified at the same time the program variable it represents

POs with model

For the post-condition of each method which is using progVar :

- Jack adds the condition :
  $\exists$ v, v $=$ P(progVar)
- Jack replaces each occurence of modVar in the post-condition of the method by v

We can link naturally a IntList model variable with a variable of the program

Quicksort with model (1)

```
//@ model IntList list ;
//@ represents list ← IntList.toList(tab) ;

/*@ public native class IntList {
  @     public static native IntList create() ;
  @     public static native IntList cons(int i, IntList c) ;
  @     public static native IntList app(IntList c1, IntList c2) ;
  @     public native boolean is_in(int i) ;
  @     public native boolean equals(IntList i) ;
  @     public native int length() ;
  @     public static native IntList toList(int []tab) ;
  @ }
  @*/
```

Quicksort with model (2)

```
/*@ requires (tab != null) && (0 <= i) && (i < tab.length)
  @          && (0 <= j) && (j < tab.length) ;
  @ modifies tab[i], tab[j], list ;
  @ ensures tab[i] == \old(tab[j]) && (tab[j] == \old(tab[i])) &&
  @          list.permutation( \old(list)) ;
  @*/
  public void swap(int i, int j) {
      int tmp ;
      tmp = tab[i] ;
      tab[i] = tab[j] ;
      tab[j] = tmp ;
  }
```

Conclusion

- The quicksort was annotated in different ways, and a proof has been made in its entirety for the version without the predicates
- We can now use some Coq predicates in Jack's JML annotations
- The use of Coq predicates for model variables is still in developpement
  $\longrightarrow$ We have done some refinement on the datatype level
- The next step would be to do it more on the program level
  $\longrightarrow$ Maybe use the JML's model program construct

Download links

Jack :

http://www-sop.inria.fr/everest/soft/Jack/

CoqEditor (only works with the CVS version of Coq !) :

http://www-sop.inria.fr/everest/soft/Jack/UpdateSite/

eassert, quicksort :

http://www-sop.inria.fr/everest/personnel/Julien.Charles/