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
- It is still a work in progress
  (the proof not using the predicates is finished, not the other)

General Plan

1. Basics
2. Annotating in a normal way
3. Using purity
4. Using ghost and model variables

Basics

1. The Java Modelling Language (JML)
2. Jack
3. Coq

JML

The Java Modelling Language (JML) is used to annotate the Java programs we want to verify with Jack.
In JML you can express loop invariants, assertions method's pre and post conditions.
JML has a really large syntax so every tool (except maybe JMLrac) implements only a portion of its syntax.

JML constructions used (1)

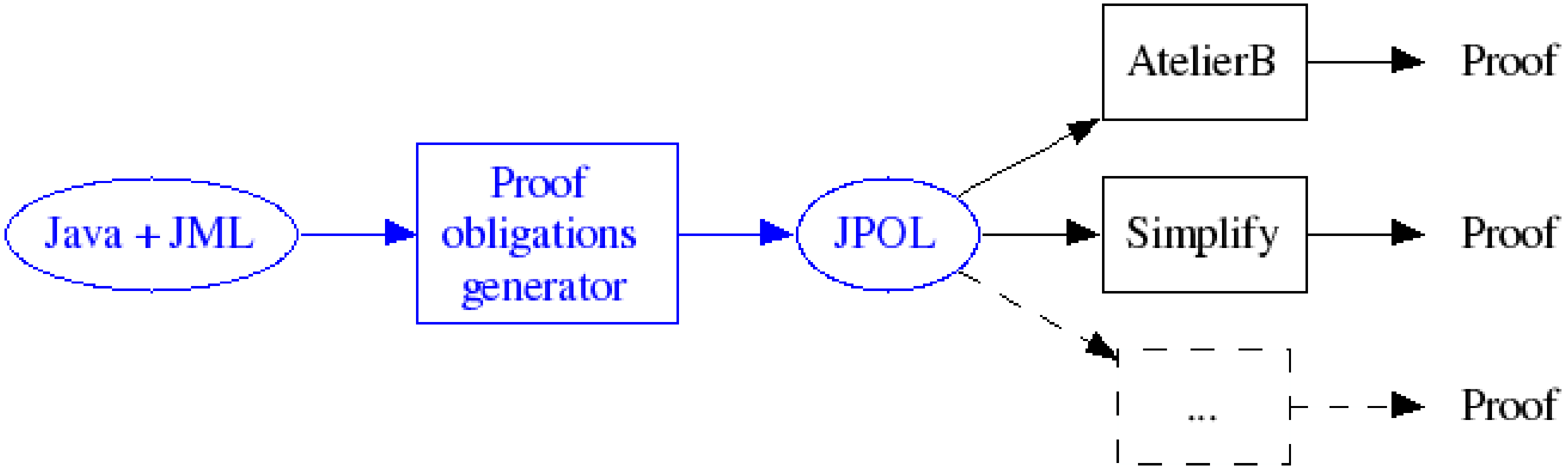
- Expressions used in annotations are special boolean expressions :
  - \forall
  - \exists
  - \old
  - methods if they are pure
- Methods :
  - requires
  - ensures
  - modifies
- Assertions with the assert construct

JML constructions used (2)

- Loop invariants, termination :
  - loop_invariant
  - decreases
  - loop_modifies
- ghost variables
  - ghost v ;
  - set v = value ;

Jack (1)

- Programmed mainly by Lilian Burdy
- Uses a weakest precondition calculus to generate proof obligations
- The proof obligations are decomposed w.r.t. the different execution cases



Jack (2)

- At first single prover (B)
- Multi-prover interface (Simplify, Coq, PVS)
- 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

Coq is based on the calculus of inductive constructions.

You can express : types, axioms, functions variables, definitions, lemmas...

Lemmas have to be proved.

To prove a lemma you have to build a proof term out of the types, axioms, variables...

To build this proof term you use special commands called tactics within a proof script.

Custom tactics can be created composing more primitive ones.

It can easily express the logics for 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.

Annotating in a normal way

1. The QuickSort algorithm
2. Kind of proof obligations
3. Annotations
4. A tactic : eassert

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

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

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) ;
  @*/
```

A tactic : eassert (1)

Many of the goals in this proofs were with split inequalities

- The hypothesis were of the form :

  H : $\forall$ v, A $\rightarrow$ B(v)... $\rightarrow$ a $<$ b

  H' : $\forall$ v, A' $\rightarrow$ B'(v)... $\rightarrow$ b $<$ c

- the lemma to prove :

  a $<$ c

- I had troubles directly instanciating the hypothesis to have :

  a $<$ b and b $<$ c.

$\rightarrow$ the tactic eassert

eassert (2)

The tactic is like assert, but it permits holes when instanciating the hypothesis, and the holes are turn into subgoals.

For a goal like :

H : A $\rightarrow$ B $\rightarrow$ C

==========================

G

eassert(h := H _ _).

turns it into 3 goals :

- One in order to prove A
- One in order to prove B
- One still asking to prove G but with an hypothesis added (h : C)

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 them to be deterministic The main problem : keeping track of the pure method name

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.
```

Simplifying pure methods

This method of defining pure methods :

- Useful for the readability of the lemma,
- Bad for the proof

$\longrightarrow$ tactics unfold_pure, unfold_pure_all are generated to directly unfold the definitions in a specified hypothesis or for all the lemma

Hypothesis of the form :

H : $\forall$ result, meth_pure = result $\longrightarrow$ result = true

are turned to

H : meth_pure = true

with a tactic called simpl_pure.

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);
}
```

- Go further in this direction :

  directly defining the pure method in Coq

Coq predicates in JML

How the predicates are defined for Jack :

- Within JML :

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

- In the file userTactics.v :

  Definition withinBounds :

  (Reference $\rightarrow$ t_int $\rightarrow$ Reference) $\rightarrow$ Reference $\rightarrow$ t_int $\rightarrow$ bool :=

  fun intelements tab value =>

  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 Coq__is_sorted(int[] tab, int beg, int end);
  @ public native boolean Coq__is_inf(int [] tab, int value, int beg, int end);
  @ public native boolean Coq__is_sup(int [] tab, int value, int beg, int end);
  @ public native boolean Coq__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 (but not enough)

→ It can facilitate the proofs or at least their understanding

Using ghost and model variables

1. Ghost variables
2. A quicksort annotated with ghosts
3. Model variables
4. Quicksort with model
5. A new tactic

Ghost variables

- Ghost variables are specification variables.
- They are declared through the keyword ghost :
  //@ ghost MyClass i ;
- They can be used within specifications
- Their value can be modified with the set instruction :
  //@ set i $\leftarrow$ MyClass.pureFactory() ;
  The set instruction allows only JML expressions and pure methods

Ghost and predicates

In Jack ghosts variables are treated like program variables
(the set instruction being the equivalent to an assignement).
Adaptation to the predicates :

- In the Coq plugin, a flag is added to define ghost natively, its definition being made by a user (in userTactics.v) :
  Variable i : Z.
- With the Coq predicates we can define a native pure Coq method :
  in the java program :
  //@ public static native MyClass pureFactory() ;
  in Coq :
  Definition pureFactory := 3.
  → We can use ghost data structure entirely defined within Coq !

Modelling with ghosts

Ghost variables can model the behaviour of program variables :

- if the assignments are made at the same steps
- if there is a transformation function between the modelled variable and the ghost variable

For the quicksort we decided to model the array with a list library, entirely written in Coq to use in specifications

A quicksort annotated with ghosts (1)

We change the field declaration to :

```
private int [] tab;
//@ ghost IntList list;
```

```
public class IntList {
    /*@ public static native IntList Coq__create();
      @ public static native IntList Coq__cons(int i, IntList c);
      @ public static native IntList Coq__app(IntList c1, IntList c2);
      @ public native boolean Coq__is_in(int i);
      @ public native boolean Coq__equals(IntList i);
      @ public native int Coq__length();
      @ public static native IntList Coq__toList(int []tab);
      @*/
}
```

A quicksort annotated with ghosts (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.equals(IntList.toList(tab)) && list.permutation( \old(list));
  @*/
  public void swap(int i, int j) {
      int tmp;
      tmp = tab[i];
      tab[i] = tab[j];
      tab[j] = tmp;
      //@ set list = IntList.toList(tab);
  }
```

Model variables

- Model variables are ghost 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

With ghost variables, to simulate this behaviour, we had to add to the specification of the methods this equality by hand
$\longrightarrow$ Better correctness than with ghost variables

A new tactic

This gives these kind of proof obligations in Coq :

(some_hypothesis) $\longrightarrow$

$\exists$ v, v $=$ P(progVar) $\wedge$ post

- Usually we have to introduce v as an existential variable, and then find the value of v which is in fact the result P(progVar).
- Problem : P is a function while v isn't (the reflexivity is not trivial)
  $\longrightarrow$ We have added a tactic, refl_goal to solve these kind of problems

Quicksort with model (1)

The only modifications concerns the variable declaration :

//@ ghost IntList list ;

is replaced by :

//@ model IntList list ;

//@ represents list $\leftarrow$ IntList.toList(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 encouraging first results
- Correctness of the different approaches
- Adapting already done proofs using model variables (refinement) ?

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/`