Implementation of the proof transformer

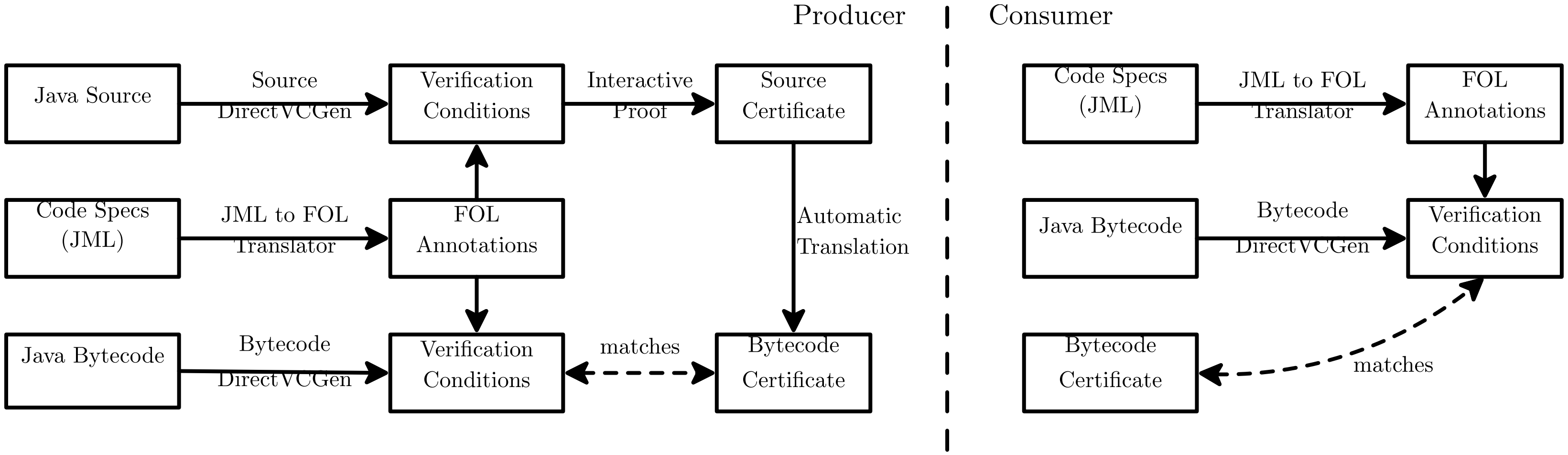
Julien Charles
INRIA Sophia-Antipolis, France

EVEREST

Thursday, june the 19th 2008

What did we do ?

- Reproduce the Proof Compiling Compiler to reality



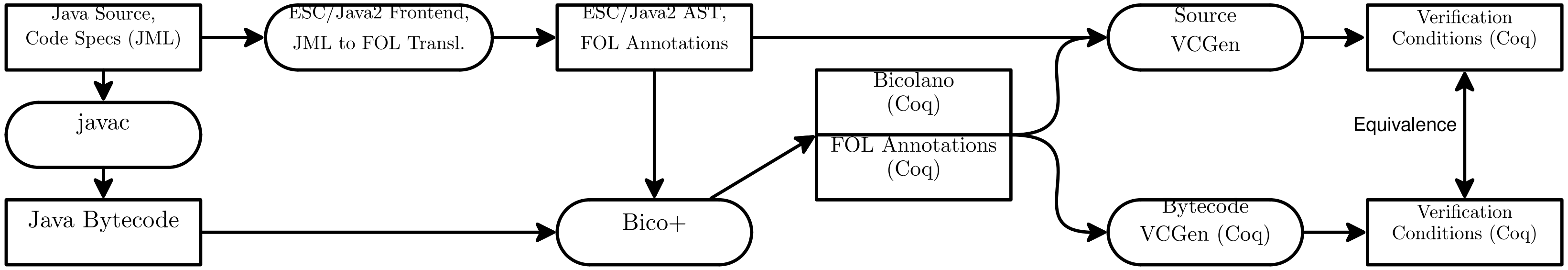
(ok, in a *direct* way)

- Uses extensively Coq

General Plan

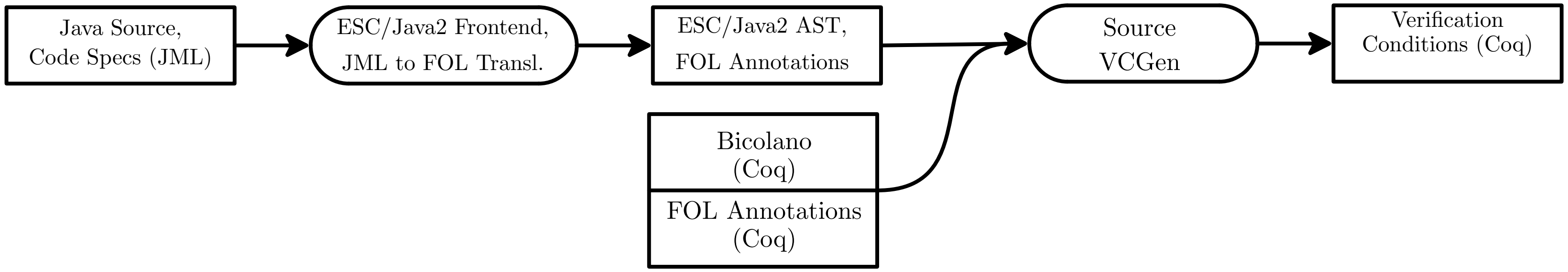
1. DirectVCGen Source
2. DirectVCGen ByteCode
3. Outputs (Bill)

General Plan cont.



1. The DirectVCGen on Source

DirectVCGen Src



- Uses a weakest-precondition calculus
- At the end of the chain background predicates
  $\longrightarrow$ It has been out-sourced to Bico

JML to FOL

...JML formulas are parsed by ESC/Java2 (JavaFE) and translated to FOL.

Hermann will present some details of it in his talk.

A glimpse of wp

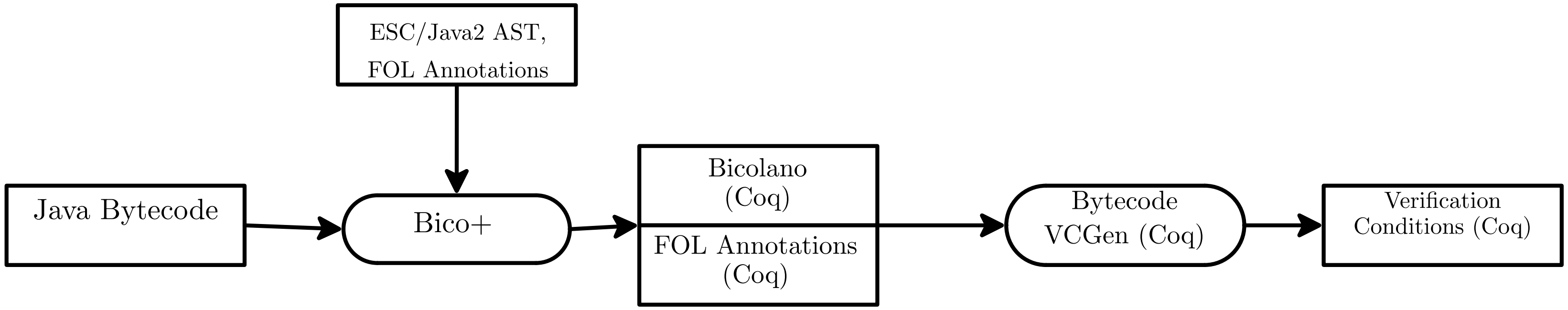
- Pretty standard
- Handles all of Java construct except the switch
- Some rules :

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{assert}\ \phi,\ \ \psi) = \phi \wedge \psi, \emptyset}$$

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{set}\ v,\ \psi) = \forall v.\psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{set}\ v\ val,\ \ \psi) = \psi\{v \mapsto val\}, \emptyset}$$

2. The DirectVCGen on Byte Code

DirectVCGen Bc



- 2 main parts
- Wp in Coq is a contribution of Benjamin Grégoire and Jorge Sacchini
- It had its generation process to be automatized

Bico^{+}

- Bico : Java bc to bicolano
- At first an effort from Warsaw
- It $adds$ specifications, modularity, automations
  $\longrightarrow$ An enhancement of Bico with annotations.

Generation

It generates an output compatible with the DirectVCGen Bytecode :

```
Module ClassNameAnnotations.
  Module MethodName.
    ...
    Definition global_spec : GlobalMethSpec := (pre ,, post).
    ...
    Definition assertion := ...
    Definition assumption := ...
    Definition local_spec : LocMethSpec := (assertion ,, assumption).
    Definition spec :=
      ((33%nat,,global_spec),,local_spec).
  End MethodName.
End ClassNameAnnotations.
```

Annotation table

- Annotations are turned into Coq definitions that can be used both for the source verification and for the bytecode verification
- Form of an assertion (and invariant) :
  `fun pre_heap pre_vars heap vars => P.`
- It permits to have more readable vcs

```
Definition mk_invariant0:=
   fun (_pre_heap: Heap.t) (_pre_n_28_40: Int.t) (heap: Heap.t) (this: value) (n_28_40: Int.t) (i_29_8: Int.t) =>
   ((i_29_8 <= (n_28_40 + 1)) /\
    ((do_hget heap (Heap.DynamicField this BillSignature.sumFieldSignature)) <=
       ((do_hget _pre_heap (Heap.DynamicField this BillSignature.sumFieldSignature)) +
         (((i_29_8 + 1) * i_29_8) / 2)))).
 Definition invariant0 (s0:InitState) (s:LocalState): list Prop :=
   (let _pre_heap := (snd s0) in
   let _pre_n_28_40 := (do_lvget (fst s0) 1%N) in
   let heap :=  (fst (fst s)) in
   let this := (do_lvget (snd s) 0%N) in
   let n_28_40 := (do_lvget (snd s) 1%N) in
   let i_29_8 := (do_lvget (snd s) 2%N) in
     mk_invariant0 _pre_heap _pre_n_28_40 heap this n_28_40 i_29_8):: nil.
```

Bico and loops (The loop problem)

- Source to Bytecode :
  $\longrightarrow$ Annotations are hard to translate
- What is the right way ?
- Usually a simple loop is compiled this way :

```
    ... // initialisation instrs.
    goto test
body :
    ...
test :
    ...
    ifcmp body
```

- Further enhancements are needed

3. Outputs and more

What is generated

Play with Bill

Bill Program (D.3.1) :

```
abstract class Bill {
  /*@ public invariant 0 <= sum; @*/
  ...
  //@ requires 0 < n;
  //@ ensures sum <=  \old(sum) + n*(n+1)/2;
  public boolean produce_bill(final int n) {
     ...
      //@ loop_invariant i <= n + 1 && sum <=  \old(sum)+(i+1)*i/2;
      for (i = 1; i <= n; i++) {
        sum = sum + round_cost(i);
      }
    ...
  }
```

An overview of the VC

- Source VC :

```
(forall (_pre_this:value), (forall
 (_pre_n_28_40:Int.t), (forall (_pre_i_29_8:Int.t), (forall
 (_pre_heap:Heap.t), (forall (this:value), (forall (n_28_40:Int.t),
 (forall (i_29_8:Int.t), (forall (heap:Heap.t),
 ((BillAnnotations.produce_bill.mk_invariant0 _pre_heap _pre_n_28_40
 heap this n_28_40 i_29_8) -> (((i_29_8 <= n_28_40) -> (((this =
 Null) -> (forall (heap3:Heap.t), (forall (x20:value), (((Heap.new
 heap p (Heap.LocationObject
 java_lang_NullPointerExceptionType.name)) = (Some ((loc x20) ,
 heap3))) -> ((Int.range 0) -> ((assign_compatible p heap3 (Num (I
 (Int.const (0)))) (PrimitiveType BOOLEAN)) -> ((compat_ValKind_value
 Ival (Num (I (Int.const (0))))) ->
 (BillAnnotations.produce_bill.mk_post (Normal (Some (Num (I
 (Int.const (0)))))) _pre_heap heap3 _pre_this _pre_n_28_40)))))))
 /\ ((not (this = Null)) -> (((this = Null) -> (forall...
```

- Bytecode VC :

```
interp_swp BicoAnnotations.anno_prog BicoProgram.program
    (certifiedMethod BicoAnnotations.anno_prog BillSignature.produce_bill Bill.produce_bill
                    BillAnnotations.produce_bill.spec).
```

More on the vcs

- Reduces to simple proof obligations :

```
_pre_this : value
_pre_n_28_40 : Int.t
_pre_i_29_8 : Int.t
h : Heap.t
this : value
n_28_40 : Int.t
i_29_8 : Int.t
h0 : Heap.t
H : BillAnnotations.produce_bill.mk_invariant0 h _pre_n_28_40 h0 this n_28_40
      i_29_8
H0 : i_29_8 <= n_28_40
H1 : this = Null
h1 : Heap.t
x20 : value
H2 : Heap.new h0 p (Heap.LocationObject
          java_lang_NullPointerExceptionType.name) =  Some (loc x20,, h1)
H3 : Int.range 0
H4 : assign_compatible p h1 (Num (I 0)) (PrimitiveType BOOLEAN)
H5 : compat_ValKind_value Ival (Num (I 0))
H6 : assign_compatible p h1 _pre_this
       (ReferenceType (ClassType BillType.name)) /\ 0 < _pre_n_28_40
______________________________________(1/10)
do_hget h1 (Heap.DynamicField _pre_this sumFieldSignature) <=
(do_hget h (Heap.DynamicField _pre_this sumFieldSignature)) +
  ((_pre_n_28_40 * (_pre_n_28_40 + 1)) / 2)
```

The equivalence relation

- An equivalence vc is also generated, of the form :

```
Lemma l:
    vc_source <-> vc_bytecode.
```

- It is proved automagically by a group of tactics.
- The algorithm is the following :

  It syntactically make the two vcs correspond :

  $$\frac{\text{H : P} \;\rightarrow \text{P'}}{\text{P} \;\rightarrow \text{P'}} \qquad \frac{\text{H : P'}}{\text{P'}}$$

  It instanciates the local variables using their orders to determine their numbers

  It removes the operands stack

  It introduces the heap

The main tactic

```
Ltac magickal :=
   repeat match goal with
   | [ |- forall lv: LocalVar.t, _ ] =>let lv := fresh "lv" in
                                         intro lv; mklvget lv 0%N
   | [ H: forall lv: LocalVar.t, _ |- _ ] => mklvupd MDom.LocalVar.empty 0%N
   | [ |- forall os: OperandStack.t, _ ] => intro
   | [ H: forall os: OperandStack.t, _ |- _ ] =>
              let H' := fresh "H" in (assert (H' := H OperandStack.empty); clear H)
   | [ H : forall y: Heap.t, _ |- forall x: Heap.t, _] =>
              let x := fresh "h" in
              (intro x; let H1 := fresh "H" in (assert (H1 := H x);
               clear H; try (clear x)))
   | [ H : forall y: Int.t, _ |- forall x: Int.t, _] =>
              let x := fresh "i" in (intro x; let H1 := fresh "H" in
                    (assert (H1 := H x); clear H; try (clear x)))
   | [ H : _ -> _ |- _ -> _] =>let A := fresh "H" in  (intros A; let H1 := fresh "H" in (assert (H1 := H A); clear H; clear A))
   | [ H : _ /\ _ |- _ /\ _] =>let A := fresh "H" in
                                      let B := fresh "H" in
                                      (destruct H as (A, B); split; [clear B | clear A])
    end.
```

Conclusion

- The DirectVCGen :
  - it permits to use proofs done on bytecode vcs to proofs on sourcecode vcs
  - it is fully automated
  - Bico has turned into Bico^{+}
- Future work :
  - More tests
  - More user-friendly
  - Optimizations ?