ESC/Java Approach

Wishnu Prasetya

wishnu@cs.uu.nl
www.cs.uu.nl/docs/vakken/pv

ESC/Java

- *Extended Static Checker for Java* → an implementation of Hoare Logic.

- Semi-automatic → theorem prover back-end.

  It is not intended to verify complex functional specification. Instead, the aim is to make your static checking more powerful.

- Spec# is something similar, but for C#. The base-language is called Boogie → reusable core.

```java
1:  class Bag {
2:      /*@ non_null */ int[] a;
3:      int n;    //@ invariant 0 <= n & n <= a.lenght
4:
5:      Bag(int[] input) {
6:        n = input.length;
7:        a = new int[n];
8:        System.arraycopy(input, 0, a, 0, n);
9:      }
10:
11:     int extractMin() { //@ requires n >= 1 ;
12:       int m = Integer.MAX_VALUE;
13:       int mindex = 0;
14:       for (int i = 1; i <= n; i++) {
15:         if (a[i] < m) {
16:           mindex = i;
17:           m = a[i];
18:         }
19:       }
20:       n--;
21:       a[mindex] = a[n];
22:       return m;
23:     }
```

```
Bag.java:15: Warning: Possible null dereference
      if (a[i] < m) {
           ^
Bag.java:15: Warning: Array index possibly too large
      if (a[i] < m) {
           ^
Bag.java:21: Warning: Possible null dereference
    a[mindex] = a[n];
                 ^
Bag.java:21: Warning: Possible negative array index
    a[mindex] = a[n];
                 ^
```

```
class Bag {
    int[] a;   /*  @ non_null   */
    int n;     //  @ invariant 0 ≤ n & n ≤  a.length

    Bag (int[] input) {
        n = input.length;
        a = new int[n];
        System.arraycopy(input, 0, a, 0, n);
    }

    int extractMin() {   //@ requires n ≥ 1
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 1; i <= n; i++) {
            if (a[i] < m) {
                mindex = i;
                m = a[i];
            }
        }
        n--;
        a[mindex] = a[n];
        return m;
    }
}
```

Possible null deref.

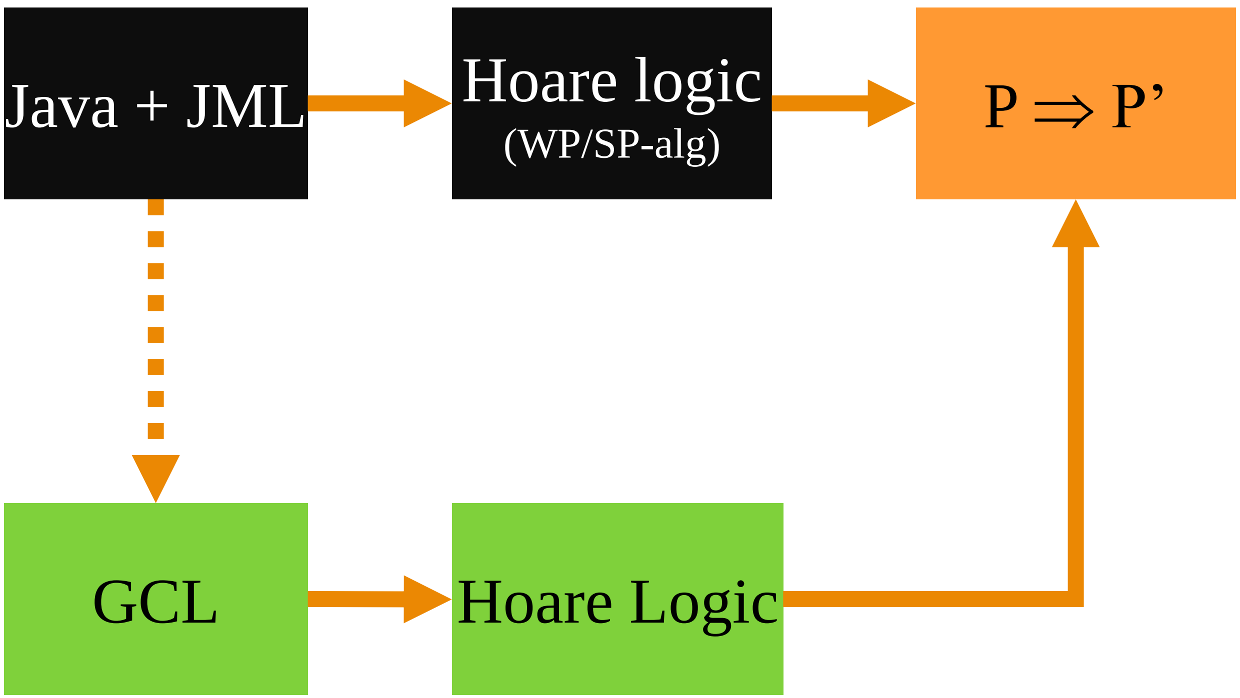
Index possibly too large

Still persist despite the inv.
→ real bug

Index possibly negative

Architecture ESC/Java

Implementing the Hoare logic to work directly on Java is complex and error prone; but in theory you'll get better error messages.



ESC/Java first render Java to a much simpler lang. GCL. The Hoare logic operates on GCL.

In principle this core is reusable. Alternatively, you can use the Boogie core.

Guarded Command Language (GCL)

- *cmd* →

  var = *expr*
  | **skip**
  | **raise** // throw an exception
  | **assert** *expr*
  | **assume** *expr*
  | **var** *variable*$^+$ **in** *cmd* **end** // locvar with scope
  | *cmd* ; *cmd*
  | *cmd* ! *cmd* // try-catch
  | *cmd* [] *cmd* // non-determ. choice
- *expr* : formula or term from untyped first-order pred. Logic
- Also of the form *Label x e* → to tag *e* with feedback information
- Data type : bool, int, infinite arrays

Non-termination, Abortion, Exception

- A state of an a GCL program has an additional flag:
  - *Normal*
  - *Exceptional*

    This is set by **raise**, and unset upon entering the handler in C!D.
  - *Error*

    This is set by violating assert; cannot be unset.

We first extend "post-condition"

- 'post-condition' is now a <u>triple</u> :

  (N , X , W)

  These are predicates,

  N : post-cond if C terminates in a normal state
  X : post-cond if C terminates in an exceptional state
  W : post-cond if C terminates in an error state.

- Example:

{ x>0 } assert i>0 ; a[i]:=x { a[i]>0, false, i≤0 /\ x>0 }

The logic is based on pre-algorihm

- **pre** = "sufficient pre-condition"

  But we also see it as a *predicate transformation algorithm*:

  $$\mathbf{pre} : \text{Statement} \rightarrow \text{Predicate} \rightarrow \text{Predicate}$$

  such that:

  $\boxed{\{\ \mathbf{pre}\ S\ Q\ \}\quad S\quad \{\ Q\ \}}$ is always valid.

Variations of the concept "pre"

- **wp** (weakest pre-condition)

  Is a predicate transformer that constructs the weakest pre-condition such that S terminates in Q.

- **wlp** (weakest liberal pre-condition)

  As wp, except that it does not care whether or not S should terminate.

- We will now give you the explicit definition of wlp for GCL…

WLP

- { ? } **skip** { x=0, y=0, z=0 }

$$\text{wlp } \textbf{skip } (N,\_,\_) = N$$

- { ? } x:=e { x=0, y=1, z=2 }

evaluating *e* is assumed not to abort (as in uPL).

$$\text{wlp } (x = e)\ (N,\_,\_) = N[e/x]$$

WLP

- { ? } **raise** { x=0, y=0, z=0 }

$$\text{wlp}\ \ \textbf{raise}\ \ (\_,X,\_) \ = \ X$$

- { ? } **assert** P { x=0, y=0, z=0 }

$$\text{wlp}\ \ (\textbf{assert}\ \ P)\ \ (N,\_,X) \ = \ (P \land N) \lor (\neg P \land X)$$

- { ? } **assume** P { x=0, y=0, z=0 }

$$\text{wlp}\ \ (\textbf{assume}\ \ P)\ \ (N,\_,\_) \ = \ P \Rightarrow N$$

How Esc/Java uses these …

- u = v.x   // line 10

This would require that v is not null.

- First insert :

```
check  NullDeref@10 ,  v != null  ;
u = v.x
```

- Then desugar “check”, e.g. to (useful for error reporting!):

```
assert (Label NullDeref@10  v!=null) ;      // treat as error
u = v.x
```

- Or to :

```
assume (v!=null) ;                      // pretend it’s ok
u = v.x
```

WLP, Composite Structures

- C [] D *non-deterministically* chooses C or D.

- { ? } C [] D { N, X, W }

$$\frac{\{ P_1 \}\ C\ \{ N, X, W \} \qquad \{ P_2 \}\ D\ \{ N, X, W \}}{\{ P_1 \wedge P_2 \}\ C\ []\ D\ \{ N, X, W \}}$$

wlp (C [] D) (N,X,W)

= **wlp** C (N,X,W) $\wedge$ **wlp** D (N,X,W)

Traditional if-then

- **if** *g* **then** *S* is just **if** *g* **then** *S* **else** skip

- **if** *g* **then** *S* **else** *T* can be encoded as follows:

  assume *g* ; *S*
  []
  assume $\neg$g ; *T*

WLP, Composite Structures

- { ? } C ; { M } D { x=0, y=0, z=0 }

$$\frac{\{P\}\ C\ \{M, X, W\} \qquad \{M\}\ D\ \{N, X, W\}}{\{P\}\ C;D\ \{N, X, W\}}$$

wlp (**C ; D**) (N,X,W)

= **wlp** C (**wlp** D (N,X,W) , X , W)

WLP, Composite Structures

- C ! D executes C, if it throws an exception it then jumps to the handler D.
- { ? } C ! { M } D { N, X, W }

$$\frac{\{P\}\ C\ \{N, M, W\} \qquad \{M\}\ D\ \{N, X, W\}}{\{P\}\ C!D\ \{N, X, W\}}$$

wlp (C ! D) (N,X,W)

= **wlp** C (N , **wlp** D (N,X,W) , W)

Local Block

- **var** x **in** C **end**

  Introduce a *local* variable x, *uninitialized* → can be of any value. Any x in C now binds to this x.
- Let's do this in ordinary Hoare logic first:
- { ? } **var** x **in** assume x>0 ; y:=x **end** { y>z /\ x=0 }
- $\boxed{\textbf{wlp}\ \ (\textbf{var}\ x'\ \textbf{in}\ C\ \textbf{end})\ \ Q\ \ = (\forall x' :: \textbf{wlp}\ C\ Q)}$

(assuming fresh x'… else you need to apply subst on Q to protect refrence to x' there, then reverse the substituton again as you are exiting the block)

Back in ESC/Java logic

- Assume fresh local-vars:

{ ? } **var** x' **in** C **end** { N, X, W }

wlp (**var** x' **in** C **end**) (N,X,W)

=

(∀x':: **wlp** C (N,X,W))

How to handle program call

- You will have to inline it. Issue: how to handle recursion? → we'll not go into this.
- If a specification is available:

  { $x \geq 0$ } $P(x)$ { $\text{return}^2 = x$ } // non-deterministic!

  we can replace z := **call** $P(e)$ with :

  assert e≥0 ; **var** ret **in** { **assume** $\text{ret}^2 = e$; z := ret }

- This assumes x is passed-by-value, and P does not modify a global variable. Else the needed logic becomes quite complicated.

Handling loop

- To handle a loop, Hoare logic requires you to come up with an *invariant* .
- Option 1 : manually annotate each loop with an invariant.
- Option 2 : try to infer the invariant?
  - Undecidable problem.
  - There are heuristics, for example replacing lower/upper bounds in the post-condition with the loop counter.
    → limited strength.
- Note: ESC/Java does not have a **while** construct. Instead it has:

  loop C **end**

  This loops forever, unless it throws an exeception. Traditional loops can be encoded in this form.

Verifying annotated loop

- { ? } **while** *g* **inv** ***I*** **do** *S* { *Q* }

- Full verification :
  - Take *I* as the wlp of the loop
  - Additionally generate two verification conditions (VCs) of the loop-rule:

$$\{ I \land g \}\ S\ \{ I \} \quad \text{and} \quad I \land \neg g \Rightarrow Q$$

- Rather than explicitly generating VCs we can also encode the verifcation as:

{ ? } **assert *I* ;**
  var *v*1,*v*2,... ;
  *x*1=*v*1; *x*2=*v*2 ; ... // all variables written by the loop
  if *g* **then** { assume *I* ; S ; assert *I* ; assume false }
    else assume *I* ;
{ *Q* }

“Idempotent” loop’s post-cond

- It is a post-condition that is also an invariant. That is, it satisfies $\{ I \wedge g \}\ S\ \{ I \}$:

$$\boxed{\{\ ?\ \} \quad \textbf{while}\ g\ \textbf{do}\ i{+}{+} \quad \{\ k=0\ \}}$$

$$\boxed{\{\ ?\ \} \quad \textbf{while}\ g\ \textbf{do}\ i{+}{+} \quad \{\ i\geq 0\ \}}$$

- Then the post-condition itself is can be “used” as the wlp (it is sufficient, though may not be the weakest).

Partial logic for loop

- We only verify up to *k* number of iterations.
- This is obviously incomplete, but any violation found is still a real error → no false positives.
- Claimed to already reveal many errors.

Partial logic for loop

- We only verify up to *k* number of iterations. This is obviously incomplete, but any violation found is still a real error → *no false positives*. Claimed to already reveal many errors.
- { ? } **while** *g* **do** *S* { *Q* }

  is now transformed to:

  { ? } **if** *g* **then** { *S* ; **if** g **then assume** false } { *Q* }

- The wlp of this corresponds to doing at most 1 iteration.
- We can unroll the loop more times, e.g. up to 2 iterations :

{ ? } **if** *g* **then**
{ *S* ; **if** *g* **then** { *S* ; **if** g **then assume** false }} { *Q* }

Logic for array assignment

- Consider this assignment:

$$\{\ ?\ \} \quad a[0] := x \quad \{\ a[0] > a[1]\ \}$$

As expected, the wp is x > a[1]. But naively applying the substitution Q[e/x] can lead to a wrong result :

$$\{\ ?\ \} \quad a[0] := x \quad \{\ a[0] > a[k]\ \}$$

You cannot just leave a[k] un-replaced by x, since *k* could be equal to 0.

Logic for array assignment

- Since at this point we don't know exactly what the value of k is :

$$\{\ ?\ \}\quad a[0] := x \quad \{\ a[0] > a[k]\ \}$$

The wp is a conditional expression:

$$a[0] = (k{=}0 \rightarrow x \mid a[k])$$

- More generally, **wp** $(a[e_1] := e_2)$ Q is :

$$Q[\ (e_3{=}e_1 \rightarrow e_2 \mid a[e_3]\ )\ /\ a[e_3]\ \ ]$$

- This assumes the array has infinite range.

How to deal with objects?

- We assume each object to have a unique ID.

  E.g. just uniquely map the object's address to an integer.

- In an OO system, objects persist in a "heap" (set of objects that live in the system at the moment) ← can get side effect!

- **Heap** is modeled by a global infinite array :

  H : *ObjectContent*[*ObjectId*] // ID → Content
  N : int // size of H

- So, if *i* is the ID of object *u*, then **H**[*i*] gives us the content of *u*.

Dealing with objects

- But, since objects have fields, we use this representation instead:

  H : *ObjectContent* [*FieldName*][*ObjectId*]

- So, if u is an object with i as ID, and x is a field of u, then:

  H[x,i]   gives the value of u.x

- Leino et al use the notation *select*(x,i).

Translating the OO syntax

- u.x := v.x + y

  is translated to: H[x,u] := H[x,v] + y

- u := new Point()

  is translated to

```
u := N ;
N++ ;
H[x,u] := 0 ;
H[y,u] := 0
```

- Note that Java's array should be treated as an object, and is not translated directly to native GCL array.

Calculating WP

- u.x := e is translated to: H[x,u] := e

$$\boxed{wp\ (u.x := e)\ Q = Q[\,((y=x \wedge v=u) \rightarrow e \mid v.y) / v.y\,]}$$

- But this explodes... replacing every v.y in Q with that conditional expression. Fortunately, most can be solved *statically*:
  - if the (compile-time) type of u is not a subtype of that of v then we know that $v \neq u$
  - field-names x and y are known statically, so the condition y=x can be checked statically too.
  - extending type checking?

Calculating WP

- u := new Point() is translated to

```
u := N ;
N++ ;
H[x,u] := 0 ;
H[y,u] := 0
```

wp (u := new C) Q
=
Q [((z=x /\ v=u) $\rightarrow$ 0 | v.z) / v.z , ((z=y /\ v=u) $\rightarrow$ 0 | v.z) / v.z]

- But... u gets a new object; so for any expression v which is not syntactically the same as v, at this point cannot refer to this new object. In other words, v$\neq$u. So, the wp can be simplified to:

wp (u := new Point) Q = Q [0 / u.x , 0 / v.y]