**Verification Conditions**

The implementation of our wlp-functions:

|  |  |  |
| --- | --- | --- |
| **Hoare Logic** | **(without side effects)** | **Side effects** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Some explanations for our new definitions:

Reminder:

* wlp⟦c⟧Q Maps a command (c) and a postcondition (Q, an assertion) to the weakest precondition (W) that will make {W} c {Q} hold.
* ⊨{P} c {Q} iff ⊨ P →wlp⟦c⟧Q

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **command** | **Hoare** | |  | **VC** |
| assume |  |  | (¬A ∨ Q) | ⊨ P →(¬A ∨ Q) |
| assert |  |  | A → Q | ⊨ P → A → Q |
|  | A | ⊨ P → A |
| error | {P} error; {Q} |  | False | ⊨ P → False |

Example:

Let’s calculate the VC for: {P} c1; assert A; c2; {Q}

First, the **wlp** for seq:

Now, let’s look at :

We get two VCs:

For the **while command** we used its verification conditions:

|  |  |  |  |
| --- | --- | --- | --- |
| **command** | **Hoare** | **VC** |  |
| while |  | 1. ⊨ P → Inv 2. ⊨ Inv ∧ ¬b →Q   (3) VC ({Inv ∧ b} c ({Inv} ) | inv |

Verification conditions (2),(3) are indepedent of P (the precondition) and can be verified independently. Condition (1) is similiar to the wlp-conditions (⊨ P→ wlp⟦c⟧Q) so in our implementation of the wlp function we handle the inv as if it’s the wlp of the while command, with the addition of “side effects” (the independent conditions 2,3).

Formally: if (2)+(3) are valid, then iff {P} (while {inv} b do c) {Q}, meaning the wlp is simply inv.

**Functions (and How to Verify Them)**

**Basic Idea (Sharon):**

Each function has:

* Preconditions (“assume”)
* Postconditions (“assert”)

Example:

Definition func(y){

precond (A); [preconditions on input y]

c;

return ret;

postcond (B); [postconditions on output ret]

}

Verification Process:

1. Function code: {A} c; {B}
2. Verification in code:

Let’s look at some example:

{P} x ≔ func(y); {Q}

Verify:

* Assert: the preconditions hold for y.
* Then, assume the postconditions hold for x.

Note: All the functions we looked at return an output and do not perform in-place/global modifications. Should we keep it that way, or should the functions be able to modify global variables?

Note: We should ensure that we won’t “forget” preconditions unrelated to x. [Did we talk about it? It might be more compicated if we allow global modifications.]

Note: we don’t care for verification if the function doesn’t stop.

**Problems in this approach:** Weak postconditions: assume the postcond is f(x)>0, and look at: x≔f(1);y≔f(1);assert(x=y). We would like the assertion to be true, but the postcondition is not strong enough.

[I might’ve missed some other problems]

**Our addition:**

Add “logical function” object: We can ass FUNC(x) as a uninterepted function counterpart to func(x).

* Assert before declaration of f: forall x, [conditions that specify the attributed of func, using FUNC].
* Postcondition for func: ret =FUNC(input)

A few examples:

FACTORIAL:

First, the logical definition of FACTORIAL:

**Forall x, assume:** (x≤0 → FACT(x) = 1) && (x>0 → FACT(x)=x\*FACT(x-1))

Then, the implementation:

**Definition** fact(x){

If (x≤0){

ret ≔1}

Else{

ret ≔ x \* fact(x-1) [postcond: fact(x-1) = FACT(x-1)]

}

return ret;

postcond (ret = FACT(x));

};

**Our initial idea & its problems:**

Def f(x){

c;

return ret;

postcond A;

}

We defined the **Hoare Triple**: {forall x, A[f(x)/ret]} c; {A}

**Problems:**

1. f(x) in postcond:

def f(x){

skip;

ret 1;

postcond (F(x)=f(x))

}

The triple is: {forall x, F(x)=f(x)} skip; {F(x)=f(x)} and it’s always valid for any logical requirements on F.

1. FALSE postcond: Problem is not solved even if we avoid using f(x) in postcond.

**Definition** f(x){

Skip;

ret 1;

postcond (0 = 1 && ret = F(x));

}

The triple is: {forall x, 0=1 && f(x)=F(x)} skip; {0=1 && f(x)=F(x)} and it’s always valid for any logical requirements on F.

To conclude, to problem was: to verify a recursive function, we wanted the postcond to become precond when calling f, but it shouldn’t be a precond in the “first step of the induction”.

Some other functions we thought were good for debugging our ideas:

1. Paired functions (functions that call each other recuresively): for example, ODD and EVEN.
2. Recursive functions without a stopping condition.

TO DO: What Now?

1. Implement functions (sharon’s suggestion + our “Logical Function” addition)
2. Something with the automatic invariants sharon’s suggested?
3. Documentation of our verification process (i.e. how to write a strong enough inv, with specific examples of our trials).