Functional Verification with Frama-C / WP

Example: Factorial

Let us start by trying to prove the correctness of a function that calculates the factorial of a number, using a loop.

```
/*@ requires n >= 0;
  @ assigns \nothing;
  @ ensures \result == fact(n);
  @*/
int factf (int n)
{
  int f = 1, i = 1 ;

  /*@ loop invariant 1<=i<=n+1 && f == fact(i-1) ;
    @ loop assigns f, i;
    @ loop variant n+1-i;
    @*/
  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```

The function already has all the required annotations to specify its behavior and prove its total correctness. Note:

- the use of the `\result` operator to refer, in the postcondition, to the result of a function

- the use of a logic function `fact`, both in the postcondition and in the loop invariant.

The idea is that this logic function captures the standard (recursive) mathematical definition of factorial. However, it is not pre-defined, so it is up to the user to give its definition, as part of the specification.

It can be defined as follows:

```
/*@ logic integer fact (integer n) =
  @         (n == 0) ? 1 : n * fact(n-1);
  @*/
```

Note that we use for this the type `integer`, which is an idealized mathematical type, not a programming type (using `int` would also work, but the above version has the advantage of being independent from concrete machine types, and will work with all of them).

The function can be successfully verified with Frama-C / WP (you may need to use some other solver to help Alt-ergo).

However, if you try to check the safety behavior (by including the RTE guards) this will no longer be true, since the calculations involved really can overflow. There is no way to fix this, unless we are willing to limit the value of the argument `n`.

- Modify the precondition to be `@ requires 0 <= n <= 10` , and include `f <= 4000000` in the invariant. You should now be able to discharge all the VCs generated by RTE.

Modular Contract-based Verification

Example: Fact

To illustrate how contracts are used for modular verification, consider the following example of a function that *tabulates* (i.e. it computes a table of) factorial numbers, calling a `factf` function like the above.

```
/*@ requires n >= 0;
  @ ensures \result == fact(n);
  @ assigns \nothing;
  @*/
int factf (int n);

/*@ requires \valid(factable+(0..size-1)) && size>0;
  @ assigns  factable[0..size-1];
  @ ensures
  @    \forall integer a ; 0 <= a < size ==>  factable[a] == fact (a);
  @*/
void factab (int factable[], int size)
{
  int k = 0 ;

  /*@ loop invariant 0 <= k <= size &&
    @      (\forall integer a ; 0 <= a < k ==>  factable[a] == fact (a));
    @ loop assigns k, factable[0..size-1];
    @ loop variant size-k;
    @*/
  while (k < size) {
    factable[k] = factf(k) ;
```

```
    k++;
  }
}
```

Observe that it is the contract of `fact`, not its implementation, that is used to prove the correctness of `factab`. In fact, no implementation of `fact` is even required!

The successful proof of correctness of `factab` means that it will behave according to its contract, if a correct implementation of `fact` is given.

The two functions can thus be verified independently, but the correctness of `factab` must be seen as conditional: it depends on the existence of a correct implementation of `fact`.

Verification of the Functional Behavior of `partition`

Recall the definition of this function:

```
void swap(int t[], int i, int j, int start, int end);

int partition (int A[], int p, int r)
{
  int x = A[r];
  int j, i = p-1;

  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j, p, r);
```

```
    }
  swap(A,i+1,r,p,r);
  return i+1;
}
```

Verifying it implies:

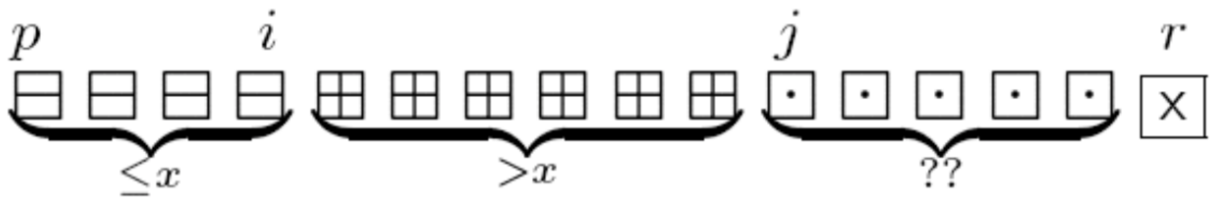
1. Checking possible runtime errors — we did this before with the help of the RTE and WP plugins
2. Writing a functional contract for `swap`
3. Understanding what `partition` does and writing its specification as a contract
4. Proving its correctness with respect to that contract. For this we need to
   a. identify an appropriate loop invariant (extending the one used before for safety verification)
5. [optionally] implementing `swap` and verifying it w.r.t. to the contract of step 2.

Observe the following postcondition. It describes the structure of the array upon termination.

It makes use of the ACSL keywords `\forall` , `\result` and `\old`, and also of the idealized `integer` type.

```
@ ensures
@      p <= \result <= r &&
@      (\forall integer l; (p <= l < \result) ==> A[l] <= A[\result]) &&
@      (\forall integer l; (\result < l <= r) ==> A[l] >  A[\result]) &&
@      A[\result] == \old(A[r]) ;
```

Here is a hint for writing a loop invariant that will allow us to prove the above postcondition. In any step of the loop we have the following structure for the array:



Thus:

```
/*@ loop invariant
  @    p <= j <= r && p-1 <= i < j &&
  @    (\forall integer k; (p <= k <= i) ==> A[k] <= x) &&
  @    (\forall integer k; (i <  k <  j) ==> A[k] >  x) &&
  @    A[r] == x;
  @*/
for (j=p; j<r; j++)
  if (A[j] <= x) {
    i++;
    swap(A,i,j,p,r);
  }
```

Now, there is a second property that should be included in the specification of the function: the elements in the array when the function exits are the same as when it was called. In other words, the array in the output state is a *permutation* of the array in the input state.

One first attempt to express that an array B is a permutation of A could be the following, using existential quantifiers:

$$\forall k : p \leq k \leq r : (\,\exists l : p \leq l \leq r : A[k] = B[l]\,)$$
$$\wedge$$
$$\forall k : p \leq k \leq r : (\,\exists l : p \leq l \leq r : B[k] = A[l]\,)$$

This is not exactly what we want: we are expressing that the *sets* of elements are the same, but we are leaving the number of *occurrences* of each element unspecified, so for instance the input could be `10 20 10 15` and the output `10 15 20 20`.

The following alternative certainly solves this problem:

$$\forall k : p \leq k \leq r : (\,\exists l : p \leq l \leq r : A[k] = B[l] \wedge A[l] = B[k]\,)$$

But another problem is raised: this version does not allow all permutations, only a restricted form, which can be described as “parallel swaps”

For instance `10 10 15 20`

- would be allowed as a permutation of `15 20 10 10`,
- but not of `10 20 10 15`

In fact the notion of permutation must be described inductively: we must define a new predicate in the logic language of Frama-C, and give derivation rules for it as follows:

```
@ inductive Permut{L1,L2}(int *arr, integer low, integer high)
{
  @  case Permut_refl{L}:
  @   \forall int *a, integer l, h; Permut{L,L}(a, l, h) ;
```

```
  @  case Permut_sym{L1,L2}:
  @    \forall int *a, integer l, h;
  @        Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h)
;
  @  case Permut_trans{L1,L2,L3}:
  @    \forall int *a, integer l, h;
  @        Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h)
  @        ==> Permut{L1,L3}(a, l, h) ;
  @  case Permut_swap{L1,L2}:
  @    \forall int *a, integer l, h, i, j, k;
  @        l <= i <= h && l <= j <= h
  @        && \at(a[i],L1) == \at(a[j],L2) && \at(a[i],L2) ==
\at(a[j],L1)
  @        && (k != i && k != j ==> \at(a[k],L2) == \at(a[k],L
1))
  @        ==> Permut{L1,L2}(a, l, h) ;
  @ }
```

We may now write a postcondition using this predicate:

```
  @ ensures  Permut{Pre,Here}(A,p,r);
```

and use the following invariant to prove it:

```
  /*@ loop invariant
    @    p <= j <= r && p-1 <= i < j &&
    @    Permut{Pre,Here}(A,p,r);
    @*/
  for(j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j,p,r);
```

```
    }
```

Using Behaviors

It is possible to group postconditions in different behaviors, which allows for improved visualization and interaction. In this example we may distinguish the permutation behavior and the partitioning behavior as follows:

```
@ behavior partition:
@ ensures
@      p <= \result <= r &&
@      (\forall int l; p <= l < \result ==> A[l] <= A[\resul
t]) &&
@      (\forall int l; \result < l <= r ==> A[l] >  A[\resul
t]) &&
@      A[\result] == \old(A[r]) ;
@ behavior permutation:
@ ensures
@      Permut{Pre,Here}(A,p,r);
```

Invariants may also be associated to behaviors as follows. Note the presence of a general part of the invariant, that is always considered, and two behavior-specific invariants.

```
@ loop invariant
@     p <= j <= r && p-1 <= i < j;
@ for partition:
@   loop invariant
@     (\forall int k; (p <= k <= i) ==> A[k] <= x) &&
@     (\forall int k; (i <  k <  j) ==> A[k] >  x) &&
@     A[r] == x;
```

```
@ for permutation:
@   loop invariant
@     Permut{Pre,Here}(A,p,r);
```

And we may then run the following:

```
> frama-c -wp -wp-bhv permutation partition_swap_complete.c
[wp] Proved goals:    3 / 3
> frama-c -wp -wp-bhv partition partition_swap_complete.c
[wp] Proved goals:    3 / 3
```

The relevant invariant will be used to establish each of the postconditions, which will also have the benefit of “thinning” the context of the corresponding VCs.

Note that many other VCs are required to prove correctness! For instance those concerning the preconditions of `swap` are generated outside these behaviors.

The complete verification of this function is achieved as follows, including RTE, frame conditions, termination, and the partitioning and permutation behaviors:

```
> frama-c -rte -wp  partition_swap_complete.c
[wp] Proved goals:   39 / 39
```

Lemmas

As always in logic, it may be useful when using WP to state and prove lemmas, that may help prove other proof obligations. For instance the following lemma states that perfuming a swap over a permuted array still results in a permutation of the original array:

```
/*@ lemma Permut_swap_sequence{L1,L2,L3}:
  @   \forall int *a, integer l, h, i, j, k;
```

```
  @   Permut{L1,L2}(a, l, h) ==>
  @     l <= i <= h && l <= j <= h &&
  @     \at(a[i],L2) == \at(a[j],L3) && \at(a[i],L3) == \at(a
[j],L2) &&
  @     (k != i && k != j ==> \at(a[k],L3) == \at(a[k],L2)) ==
>
  @   Permut{L1,L3}(a, l, h) ;
  @*/
```

A proof obligation will be created for this lemma, and the lemma will be placed in the context of subsequent proof obligations.

Exercises

1. Note that the function `factab` above performs many redundant calculations. Produce a more efficient version, noting that the next element in the table can be obtained from the previous by a simple calculation, without invoking a `fact` function.
   Prove that your improved implementation is correct with respect to the (same) contract.

2. Write a contract for a function that determines the index in an array where its maximum (or one of its maxima) is stored:

```
int maxarray(int u[], int size)
```

   a. Write an appropriate contract for it, including all annotations that are required for safe execution, and also its functional specification

   b. Write a definition for the function and prove its safety, termination, and functional correctness with respect to that contract

c. Check the following main function that invokes `maxarray`. Note the use of an `@assert` ACSL annotation after the function call to ensure that a maximum has been calculated .

```
#define LENGTH 100
int vec[LENGTH];

int maxarray(int u[], int size) { ... }

void main() {
  int max;
  max = maxarray(vec, LENGTH);

  /*@ assert 0 <= max < LENGTH &&
    @       (\forall int a; 0 <= a < LENGTH ==> vec[a] <= vec
[max]);
    @*/
}
```

3. Consider the following **Insertion Sort** algorithm.

```


/*@ predicate sorted(int *t,integer i,integer j) =
  @   ...
  @*/

/*@ requires N>0  && \valid(A+(0..N-1));
  @ assigns A[0..N-1];
  @ ensures sorted(A,0,N-1);
  @*/
void insertionSort(int A[], int N) {
```

```
    int i, j, key;
    for (j=1 ; j<N ; j++) {
        key = A[j];
        i = j-1;
        while (i>=0 && A[i] > key) {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = key;
    }
}
```

a. Complete the definition of the `sorted` predicate.

b. Insert all invariants and other annotations required to prove the total correctness of the function.

c. Modify the precondition to `N >= 0`, observe the problem that arises, and solve it.