Why3: Verification of Imperative Programs

Why3 como ferramenta de *verificação de programas*

- Linguagem de programação interna WhyML, mistura características funcionais e imperativas
- Baseada nos princípios da *Lógica de Hoare*, nas noções de pré-condição, pós-condição, e invariante de ciclo, e também no método de desenvolvimento conhecido por *design-by-contract*
- Inclui um *gerador de condições de verificação* que, dado um *programa anotado* com uma especificação, produz um conjunto de fórmulas lógicas (condições de verificação) cuja validade implicará que o programa é correcto face à sua especificação.
- As condições de verificação são provadas com recurso às múltiplas ferramentas com que o Why3 interage
- Depois de provado correcto, um programa WhyML pode ser *extraído* para uma linguagem de programação real: OCaml ou C
- É possível combinar os dois aspectos da ferramenta: a linguagem lógica, muito rica, pode ser utilizada no raciocínio sobre propriedades de programas
- Suporte para *ghost code*

Recall what an ACSL-annotated C program looks like. The (total) correctness of the following insertion sort algorithm can be established using the Frama-C WP plugin.

```
/*@ predicate sorted(int *t,integer i,integer j) =
  @   \forall integer k, integer l; i <= k < l <= j ==> t[k] <= t[l];
  @*/

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

  /*@ loop invariant
    @    1 <= j <= N && sorted{Here} (A,0,j-1);
    @ loop variant (N-j);
    @*/
    for (j=1 ; j<N ; j++) {
        key = A[j];
        i = j-1;

        /*@ loop invariant
          @    -1 <= i <= j-1 && sorted(A,0,j-1) &&
          @    (i < j-1 ==> A[j-1] <= A[j] && key < A[i+2]);
          @ loop variant i;
          @*/
        while (i>=0 && A[i] > key) {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = key;
    }
}

```

Some key points:

- Memory safety is a low-level concern in C that is conveniently addressed by both the specification language and the verification tool
- There is no logic library, so notions like “sorted” and “permutation” have to be defined / specified by users
- In this specific example we were unable to prove the permutation behavior — since the algorithm does not use a swap instruction (for the sake of efficiency), in the inner loop the array is not always a permutation of the initial array.

The first major difference when using the Why3 verifier is the existence of a very extensive logic library. For instance the section on arrays is available here: http://why3.lri.fr/stdlib/array.html

It contains in particular modules `array.IntArraySorted` and `array.ArrayPermut` that not only define the relevant notions, but also contain lemmas about them that will facilitate automated proofs.

In order to verify an algorithm with Why3 we create a module that starts by importing the required library modules, and write the algorithm in WhyML, the Why3 programming language.

```
module InsertionSort
  use int.Int
  use ref.Ref
  use array.Array
  use array.IntArraySorted
  use array.ArrayPermut

  let insertion_sort (a: array int)
    ensures { sorted a }
    ensures { permut_all (old a) a }
  =
    . . .
end
```

```
14
```

WhyML belongs to the ML family of languages, which also includes SML and OCaml.
https://en.wikipedia.org/wiki/ML_(programming_language)

It is *not* a general-purpose programming language: it is designed for verification at the algorithmic level, rather than the program level. However, Why3 offers automatic generation of both C and OCaml code from (verified) WhyML developments.

In fact, Frama-C/WP and Why3 illustrate a dichotomy between two different approaches to deductive program verification:

- on one hand we find verifiers that target specific real-world programming languages. Examples include Frama-C/WP, Verifast (for C and Java programs), KeY (for Java), or SPARK;

- on the other hand, tools like Why3 and the Microsoft Research tools Dafny / Boogie offer their own programming languages targeting verification at the algorithmic level.
  Verified real-life programs can also be obtained with these tools, either by automatic extraction, or else by encoding programs into the language of the verifier (Boogie is designed specifically to be used in this way as an intermediate verifier).

ML languages are functional, but support mutability and references. WhyML and Why3 can be used to verify both functional and imperative (or mixed functional-imperative) programs.

Let us look at the complete definition of the algorithm, following step by step the C version, but adding also the permutation behavior:

```
1  let insertion_sort (a: array int)
2    ensures { sorted a }
3    ensures { permut_all (old a) a }
4  =
5    for j = 1 to length a - 1 do
```

```
6       invariant { sorted_sub a 0 j }
7       invariant { permut_all (old a) a }
8       let key = a[j] in
9       let i = ref (j-1) in
10
11      while !i >= 0 && a[!i] > key do
12          invariant { -1 <= !i <= j-1 }
13          invariant { sorted_sub a 0 j }
14          invariant { !i = j-1 \/ (a[j-1] <= a[j] /\ key < a[!
    i+2]) }
15          invariant { permut_all (old a) a[!i+1 <- key] }
16          variant { !i }
17          a[!i+1] <- a[!i];
18          i := !i - 1
19      done;
20      a[!i+1] <- key
21    done
22
```

We make the following remarks regarding the programming language:

- As in many other languages (but not C), arrays contain length information: `length a` is the length of array `a`. Array accesses are valid within their length

- `for` loops are bounded iterations (as in Python). As such, there is no need to provide variants to establish termination, or to include trivial invariants concerning the control variables (`j` in the above example)

- A distinction is made between normal variables like `key` and `j` above (immutable, as in pure FP languages), and *references,* which offer mutability. `for` loop control variables are not references, and cannot be assigned

- The instruction `let i = ref (j-1) in ...` creates the reference `i` and initializes its contents with the current value of the expression `j-1`.
  In order to access the contents of the reference the symbol `!` is used

- Three (**!**) different assignment operators are used:
  - `=` is a binding, rather than an assignment. It is used for immutable variables, and also to initialize references (note that the reference variable itself is immutable; like a C pointer, it is its contents that can be modified)
  - `:=` is the reference assignment instruction. Think of `i := e` as the instruction `!i = e`, similar to `*i = e` in C. Thus `i := !i - 1` increments the value of `i` .
  - `←` is the array assignment operator: the instruction `a[!i+1] <- key` stores the value of `key` in position `!i+1` of the array `a`.

And regarding the specification and annotations:

- No preconditions are required regarding array allocation, as in ACSL
- The predicate `sorted` concerns the entirety of the array; `sorted_sub` expresses that a segment of an array is sorted:
  `sorted_sub a x y` means that the range between indices `x` and `y-1` is sorted.
- The expression `old a` refers to the array `a` in its initial state; `a[k <- e]` refers to the array `a` updated by setting the value of index `k` to `e`.
  This latter notation is used in the above example to express a loop invariant regarding the permutation property: as it is, the current array is not a permutation of the initial array because it does not contain the `key` element; the invariant mentions the array obtained by writing it back.

Try it Yourself: Selection sort

This lab is designed to illustrate the use of contract-based verification in Why3. You will verify a version of the selection sort algorithm relying on three different functions. Similarly to Frama-C/WP, the verification of a called function does not consider the code of the callees; instead it relies entirely on their contracts.

The **selection sort** algorithm sorts an array by successively placing in each position the "next minimum" element, as follows:

[40, 20, 10, 30, 60, 0, 80]
[**0**, 20, 10, 30, 60, 40, 80]
[**0, 10**, 20, 30, 60, 40, 80]
[**0, 10, 20**, 30, 60, 40, 80]
[**0, 10, 20, 30**, 60, 40, 80]
[**0, 10, 20, 30, 40**, 60, 80]
[**0, 10, 20, 30, 40, 60**, 80]
[**0, 10, 20, 30, 40, 60, 80**]

The array is modified by exchanging pairs of elements, using the following swap function. Note the use of the exchange library predicate to express the swapping property (we could also write this "by hand"):

```
let swap (a: array int) (i: int) (j: int) =
  requires { 0 <= i < length a /\ 0 <= j < length a }
  ensures { exchange (old a) a i j }
 let v = a[i] in
  a[i] <- a[j];
  a[j] <- v

```

Now complete the specification and invariant in the following minimum function and verify both functions `swap` and `select`.

```
let select (a: array int) (i: int) : int
  requires { 0 <= i < length a }
  ensures  { i <= result < length a }
  ensures  { . . . }
```

```
    =
    let min = ref i in
      for j = i + 1 to length a - 1 do
        invariant { . . . }
        if a[j] < a[!min] then min := j
      done;
    !min

```

Finally, complete the definition of the `selection_sort` algorithm, using the above `select` function, and verify it.

```
let selection_sort (a: array int)
   ensures { sorted a }
   ensures { permut_all (old a) a }
=
(. . .)
```

Further exercises:

1. Investigate whetheher the two behaviors are independent from each other. Is it possible to prove only the sorting or permutation behavior in an independent way?

2. Write an alternative version without a helper `select` function.

3. Replace the `swap` spec using `exchange` by your own version, including all information required for the verification of the sorting algorithm to still succeed.