Why3: Verification of Functional Program

This note is an introduction to the use of Why3 for the specific purpose of verifying functional programs (just one of the many uses of the tool). We take a tour of Why3's logic and programming languages and the ways in which they interact, and illustrate different ways to conduct inductive proofs. Finally, we also cover the refinement mechanism that is available through *module cloning*.

Insertion Sort as a logic function

We start by defining the insertion sort algorithm on integer lists as follows:

```
theory InsertionSort
  use int.Int
  use list.List
  use list.Permut
  use list.SortedInt

  function insert (i :int) (l :list int) : list int =
    match l with
    | Nil      -> Cons i Nil
    | Cons h t -> if i <= h then Cons i (Cons h t) else Cons h (insert i t)
    end

  function iSort (l :list int) : list int =
    match l with
    | Nil      -> Nil
    | Cons h t -> insert h (iSort t)
```

```
    end

```

The permutation and order properties can be expressed using predicates defined in the `list` library modules, imported as shown above. For instance the `sorted` inductive predicate is defined in the `list.Sorted` module as follows (and is in turn cloned by the `list.SortedInt` module) :

```
  inductive sorted (l: list t) =
    | Sorted_Nil:
        sorted Nil
    | Sorted_One:
        forall x: t. sorted (Cons x Nil)
    | Sorted_Two:
        forall x y: t, l: list t.
        le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y
l))

```

As defined above, `insert` and `iSort` functions are both logic functions, and we can write lemmas about them. For instance we require the following two about `insert` :

```
  lemma insert_sorted: forall a :int, l :list int.
                       sorted l -> sorted (insert a l)

  lemma insert_perm: forall x :int, l :list int.
                     permut (Cons x l) (insert x l)

```

Now, notice that `insert` has a structural recursive definition (the recursive call is performed on the tail of the list). This means that the above lemmas can both be proved using a simple induction principle; in Why3 they can be proved using the `induction_ty_lex` proof transformation.

With the above lemmas, we may now move to proving results about the sorting algorithm:

```
  lemma sort_sorted: forall l :list int.
                     sorted (iSort l)

  lemma sort_perm: forall l :list int.
                   permut l (iSort l)

```

`iSort` is also defined in a similar way, so both these lemmas can again be proved using `induction_ty_lex`.

We can state the final result about the algorithm by defining what a sorting function is, and then proving that `iSort` is such a function.

```
  predicate is_a_sorting_algorithm (f: list int -> list int) =
    forall al :list int. permut al (f al) /\ sorted (f al)

  goal insertion_sort_correct: is_a_sorting_algorithm iSort

end
```

The proof of the goal does not require induction (it results directly from the two previous lemmas).

Insertion Sort as a program function

An alternative possibility is to write the algorithm as a program (WhyML) function equipped with a *contract,* similarly to what would be done with an imperative program function.

Let us start with the helper function `insert` : its contract states that it should receive a sorted list, and it will also return a sorted list. Moreover, the result contains the same multiset of elements as the input list, extended with the inserted element.

```
module InsertionSortProgram
  use  ...

  let rec function insert (i: int) (l: list int) : list int
    requires { sorted l }
    ensures { sorted result }
    ensures { permut result (Cons i l) }
    =
    match l with
    | Nil -> Cons i Nil
    | Cons h t -> if i <= h then Cons i l else Cons h (insert
i t)
    end

```

The verification conditions generated by Why3 for this function are all easily proved with the help of an SMT solver, using one of the auto strategies (depending on your setup this may not even require splitting the VC).

There are several observations to be made here. First of all, note that whereas for a typical iterative algorithm you would have to provide one or more loop invariants that would allow for the function's contract to be established, in a recursive function the contract itself plays the role of invariant as well: the very contract of the function that is being verified is used to generate a verification condition regarding the recursive call.

A second remark is that, when compared with the logic version of the algorithm, no manual proof transformation is required now to make explicit the induction principle to be used: this is implicitly given by the function definition itself.

The same remarks apply to the `iSort` function, also proved correct using an auto strategy:

```
  let rec function iSort (l: list int) : list int
    ensures { sorted result }
```

```
    ensures { permut result l }
    =
    match l with
    | Nil -> Nil
    | Cons h t -> insert h (iSort t )
    end

  predicate is_a_sorting_algorithm (f: list int -> list int) =

    forall al :list int. permut al (f al) /\ sorted (f al)

  goal insertion_sort_correct: is_a_sorting_algorithm iSort

end
```

Observe also the final goal above, a logic statement involving the program function `iSort`. The fact that this can be written means that, in fact, `iSort` inhabits both namespaces: it is both a program function and a logic function. We remark the following:

- This is optional! We could choose to make the function exist only at the program level, in which case it would not be possible to mention it in the logic (as in the above goal)

- Only pure program functions, with no side effects, can be declared as `function` . When reasoning about functional programs it makes sense to do this, which will result in a particular kind of utilization of Why3. In practice, functional programs can be used in the logic but specified and verified using contracts, which facilities inductive proofs.
  (AND TERMINATION)

Mergesort as a program function

Let us move to a more complicated algorithm. The functional mergesort algorithm requires two helper functions, one to split the input list in two lists of the same size, and another to merge two sorted lists. We will write both as WhyML functions equipped with contracts; for `split` the contract merely says that together the result lists contain the same elements as the input; for `merge` we require the input lists to be sorted and in addition to the permutation aspect the contract states that the result is a sorted list:

```
module MergeSort
  use ...

  let rec function split (l :list int) : (list int, list int)
    ensures { let (l1,l2) = result in permut l (l1 ++ l2) }
  =
    match l with
    | Nil -> (Nil, Nil)
    | Cons x Nil -> (Cons x Nil, Nil)
    | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                             in (Cons x1 l1, Cons x2 l2)
    end

  let rec function merge (l1 l2 :list int) : list int
    requires { sorted l1 /\ sorted l2  }
    ensures { sorted result }
    ensures { permut (l1 ++ l2) result }
  =
    match l1, l2 with
    | Nil, _ -> l2
    | _, Nil -> l1
    | (Cons a1 l1'), (Cons a2 l2') -> if (a1 <= a2)

                                      then (Cons a1 (merge l1'
l2))
                                      else (Cons a2 (merge l1
```

```
l2'))
    end

```

All the VCs generated for these functions are proved automatically with SMT solvers using an auto strategy. It is worth considering this for a moment, since both functions are quite different from what we had in insertion sort and its helper function.

- The recursive call in `split` is not performed on the tail of the list, but rather on the "tail of the tail"
- `merge` on the other hand takes two argument lists; the recursive call alternatively preserves one of them, and is structural in the other argument.
- Why3 is still capable of proving the termination of both functions automatically

The main `mergesort` function implements the divide and conquer strategy using the above helpers:

```
  let rec function mergesort (l :list int)
    ensures { sorted result }
    ensures { permut l result }
    variant { length l }
  =
    match l with
    | Nil -> Nil
    | Cons x Nil -> Cons x Nil
    | _ ->   let (l1,l2) = split l in merge (mergesort l1) (me
rgesort l2)
    end

end
```

The first thing to notice is the presence of a **variant.** The termination of this

function cannot be proved automatically, and in fact Why3 will reject the definition if a variant is not provided as part of the contract.

The variant `{ length l }` will however lead to the generation of a VC that cannot be proved: it is not possible to automatically establish that `split` produces two lists that are strictly shorter than its argument.

In order to allow for termination of `mergesort` to be proved, we then add the following postcondition in the contract of `split` :

```
  ensures { let (l1,l2) = result in length l < 2 \/
            (length l >= 2 /\ length l1 < length l /\ length l2
< length l) }

```

Mergesort as a logic function

We started by seeing a logic version of insertion sort, followed by a WhyML definition of insertion sort and then of mergesort. It is only natural to ask whether the latter algorithm can also be defined in logic.

New difficulties arise, which we will take as opportunities to discuss additional features of Why3. Let us start by looking at the first helper function, `split`. It can be defined as follows:

```
  function split (l :list int) : (list int, list int) =
    match l with
    | Nil -> (Nil, Nil)
    | Cons x Nil -> (Cons x Nil, Nil)
    | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                             in (Cons x1 l1, Cons x2 l2)
    end

```

We would now like to prove the following lemma, meaning that the multiset of

elements is preserved by splitting:

```
lemma split_lm: forall l :list int.
                  let (l1,l2) = split l in permut l (l1 ++ l2)
```

However, the `induction_ty_lex` proof transformation will not work, because unlike `insert` the `split` function is not defined by simple structural recursion.

Why3 offers a way out of this difficulty in the form of a *lemma function.* This borrows from the program level of Why3 the capability to perform inductive proofs based on contracts. We will define a WhyML function that takes a list as argument, and write a postcondition corresponding to the lemma we are trying to prove (so `split` is naturally mentioned in it). The definition of the function simply expresses the induction principle that is required for the proof, by following the definition of `split`.

```
let rec lemma split_lm (l :list int) : ()
  ensures { let (l1,l2) = split l in permut l (l1 ++ l2) }
= match l with
  | Nil -> ()
  | Cons _ Nil -> ()
  | Cons _ (Cons _ l') -> split_lm l'
  end
```

The verification condition is proved, and the contract is inserted in the logic context as a lemma. The lemma function quite resembles the WhyML definition of `split` that we saw previously, but its only purpose is to provide a proof structure for its postcondition.

The same can be done for `merge` :

```
function merge (l1 l2 :list int) : list int =
  match l1, l2 with
  | Nil, _ -> l2
  | _, Nil -> l1
  | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
```

```
                                    then (Cons a1 (merge l1' l2))
                                    else (Cons a2 (merge l1 l2'))
    end

  let rec lemma merge_lm (l1 l2 :list int) : ()
    requires { sorted l1 /\ sorted l2  }
    ensures { sorted (merge l1 l2) }
    ensures { permut (l1 ++ l2) (merge l1 l2) }
  = match l1, l2 with
    | Nil, _ -> ()
    | _, Nil -> ()
    | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
                                    then merge_lm l1' l2
                                    else merge_lm l1 l2'
    end
```

One would then be tempted to write `mergesort` as the following logic function:

```
  function mergesort (l :list int) : list int
  = match l with
    | Nil -> Nil
    | Cons x Nil -> Cons x Nil
    | _ ->   let (l1,l2) = split l
                  in  merge (mergesort l1) (mergesort l2)
    end

```

This will not work: termination cannot be established automatically (because the recursive calls are not being performed on sublists of the argument `l`), so it is not possible to define mergesort as a `function` .

There is a way out of this. Recall that we know how to define mergesort as a

WhyML function that is also a logic function. What we were not able to do yet is to define it as a logic function that *is not a program function*. The difficulty is that, since automatic termination cannot be established, we would like to proved a *variant*, but variants can only be used in program functions.

This may sound a little confusing at first, but Why3 provides a way to define logic functions that are not program functions, using program constructs and contracts. Ghost functions are written as WhyML code that is only interpreted logically, not meant to be executed. Mergesort can be defined as follows in the logic namespace, with the length of the argument list used as variant:

```
let rec ghost function mergesort (l :list int) : list int
  variant { length l }
= match l with
  | Nil -> Nil
  | Cons x Nil -> Cons x Nil
  | _ ->   let (l1,l2) = split l
               in  merge (mergesort l1) (mergesort l2)
  end

```

Its correctness is established through the following lemma function, where the `assert` statements act as intermediate lemmas:

```
let rec lemma mergesort_lm (l :list int) : ()
  ensures { sorted (mergesort l) }
  ensures { permut l (mergesort l) }
  variant { length l }
=
  match l with
  | Nil -> ()
  | Cons _ Nil -> ()
  | _ ->   let (l1,l2) = split l
           in assert { permut l (l1 ++ l2) } ;
              mergesort_lm l1 ;
```

```
                mergesort_lm l2 ;
                assert { permut l (mergesort l1 ++ mergesort l
2) }
    end

```

Refinement in Why3

Consider the following alternative WhyML version of mergesort:

```
module MergeSort
  use ...

  val function split (l :list int) : (list int, list int)
    ensures { let (l1,l2) = result in length l < 2 \/
              (length l >= 2 /\ length l1 < length l /\ length
l2 < length l) }
    ensures { let (l1,l2) = result in permut l (l1 ++ l2) }


  val function merge (l1 l2 :list int) : list int
    requires { sorted l1 /\ sorted l2  }
    ensures { sorted result }
    ensures { permut (l1 ++ l2) result }


  let rec function mergesort (l :list int)
    ensures { sorted result }
    ensures { permut result l }
    variant { length l }
  = match l with
```

```
    | Nil -> Nil
    | Cons x Nil -> Cons x Nil
    | _ ->   let (l1,l2) = split l
             in merge (mergesort l1) (mergesort l2)
    end

end
```

It differs from the previous version in the fact that no definitions are given for the helper functions. Instead, they are just declared (using `val`) with a signature and contract. The contracts contain all the necessary information to allow for the correctness of `mergesort` to be proved — its verification condition can be proved in the same way as in the previous version.

If we want to provide a concrete definition of mergesort, by giving definitions of `split` and `merge`, we must now *clone* the `MergeSort` module inside a new module, as follows:

```
module MergeSortRefnm
  use ...

  let rec function split (l :list int) : (list int, list int)
    ensures { let (l1,l2) = result in length l < 2 \/
              (length l >= 2 /\ length l1 < length l /\ length
l2 < length l) }
    ensures { let (l1,l2) = result in permut l (l1 ++ l2) }
  = match l with
    | Nil -> (Nil, Nil)
    | Cons x Nil -> (Cons x Nil, Nil)
    | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                             in (Cons x1 l1, Cons x2 l2)
    end

  let rec function merge (l1 l2 :list int) : list int
```

```
    requires { sorted l1 /\ sorted l2 }
    ensures { sorted result }
    ensures { permut (l1 ++ l2) result }
    variant { length (l1 ++ l2) }
  = match l1, l2 with
    | Nil, _ -> l2
    | _, Nil -> l1
    | (Cons a1 l1'), (Cons a2 l2') -> if (a1 <= a2)

                                      then (Cons a1 (merge l1'
l2))
                                      else (Cons a2 (merge l1
l2'))
    end

  clone MergeSort with val split, val merge

  goal thisReallyWorks :
    forall l :list int. let ls = mergesort l
                            in sorted ls /\ permut ls l

end
```

Cloning the `MergeSort` module inside `MergeSortRefnm` will copy the former into the latter, instantiating the elements mentioned after `with` : the functions `split` and `merge` are given a definition in the cloning module. Observe that this will generate verification conditions to establish that the refinement of the contracts declared in `MergeSort` with the definitions of `MergeSortRefnm` is correct, i.e. for each function, the contract in the cloning module is not weaker than the contract in the cloning module.

In the present example both functions have exactly the same contract in the defined function as in the cloned declaration, so these VCs are proved trivially. Together, the VCs generated for the two modules imply that we have a correct implementation of merge sort, because:

- The validity of the `MergeSort` VCs implies that the `mergesort` implementation is correct if `split` and `merge` are implemented according to the specifications given in the module.

- The validity of the `MergeSortRefnm` VCs implies that the implementations of `split` and `merge` are correct according to the specifications given in this module, and moreover (for the VCs generated by cloning) they are also correct with respect to the contracts in the `MergeSort` module (which happen to be the same).

Exercise

Consider a (polymorphic) inductive type for (immutable) binary trees and the specification and implementation of an ordered insertion function on trees of type `int`.

```
type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)

let rec add (t : tree int) (v : int) : tree int =
    requires { sortedBT t }
    ensures  { sortedBT result }
    ensures  { size result = size t + 1 }
    ensures  { forall x : int. memt result x <-> (memt t x \/ x = v) }
    ensures  { forall x : int. num_occ x result =
                                 if x = v then 1 + num_occ x t else num_occ x t }
    match t with
     | Empty -> Node (Empty) v (Empty)
     | Node t1 x t2 ->
       if v <= x then Node (add t1 v) x t2
```

```
                    else Node t1 x (add t2 v)
    end

```

Define all the predicates and functions used in the specification and prove the correctness of `add`.

We note the following:

- The spec contains redundancy. Where?
- On deciding whether a given function/predicate should be defined in the program, logic, or both namespaces:
    i. first, bear in mind that Why3 supports code extraction from proved WhyML programs. Functions defined only in logic may not be extracted
    ii. functions defined only as programs may not be used in specifications
    iii. functions with side effects may not exist in the logic namespace
    iv. functions making use of logical notions such as quantification or mathematical equality may not exist in the program namespace
- From this we conclude:
    - `add` can be extracted, and can also be used in logic if we include the keyword `function`, since it does not change the global state
    - `sortedBT`, `size`, `memt`, and `num_occ` must be pure functions defined in the logic namespace, and depending on how they are implemented, may possibly also exist in the program namespace.

Three versions of leq_tree: logic, program, dual

```
predicate leq_tree (x : int) (t : tree int) =
  forall k : int. memt t k -> k <= x


let rec predicate leq_tree (x : int) (t : tree int)
= match t with
  | Empty -> true
  | Node t1 k t2 -> k <= x && leq_tree x t1 && leq_tree x t2
```

```
    end


  let rec predicate leq_tree (x : int) (t : tree int)
    ensures { result <-> forall k : int. not (memt t k) \/ k <= x }
  = match t with
    | Empty -> true
    | Node t1 k t2 -> k <= x && leq_tree x t1 && leq_tree x t2
    end

```