1. Verification of Functional Programs with Why3

The Why3 Platform

Why3 is targeted at deductive formal verification of algorithms and programs.

The most important conceptual components of the platform are the following:

- WhyML, a functional-imperative programming language of the ML family
- A logic language, which is used to write properties and annotations concerning the WhyML programs. These include preconditions, postconditions, loop invariants, variants, asserts ...
- A standard library containing many useful logical theories and data structures, including arithmetics, sets, lists, and maps (dictionaries).

The **Why3 tool** implements the following functionality:

- Generation of *verification conditions* (VCs): taking a program and a specification for it, the tool generates a set of first-order proof obligations that are sufficient to prove the correctness of the program: if all the VCs are valid then the program is guaranteed to be correct w.r.t its specification
- Interaction with proof tools: Why3 has interfaces allowing it to use as background every popular first-order proof tool (including SMT solvers and interactive theorem provers). The tool will send proof obligations to any tool available in a given installation
- Additionally, the Why3 IDE offers very useful functionality including syntax highlighting (very useful to associate VCs with the parts of the code from which they were generated), convenient access to *proof transformations*, and *proof management (*including storing and replaying proof sessions)

Note

> *Previously in this course the **Alloy** tool was also used for formal verification. However, in Alloy the word* verification *is used as a synonym for* model checking *(algorithmic state space exploration), whereas in Why3 it is based on actual deductive proofs.*
> *Moreover, Alloy targets systems specified at an abstract level, whereas Why3 is used to verify algorithms.*

The Logic Language and Why3 as a Proof Tool

Although this was probably not the original priority, Why3 has become popular as a proof tool on its own (its logic functionality is also accessible through an API).

This is due to the extremely convenient interfacing with external proof tools, and also to the features of the logic language, which extends standard FOL with a set of features that are highly convenient for applications in computer science.

The logic language is *first-order* and *many-sorted*.
It is important to clarify the difference in the meaning of quantification in **Alloy** and in many-sorted logic.

Consider the formula $\forall x : A.\ \varphi$.

- In Alloy, which is untyped, A is a predicate and the 'typing' is just syntactic sugar:
  $\forall x : A.\ \varphi \equiv \forall x.\ A(x) \rightarrow \varphi$

- Whereas in the typed setting (and Why3's logic),
  $\forall x : A$ denotes quantification over inhabitants of type A

The Why3 logic includes:

- *user-defined types*, including algebraic (inductive), and polymorphic types. Equality predicates on algebraic types are automatically available.

- defined predicates and functions, including inductive and polymorphic definitions

- *inductive predicates*, defined in a closed way by a set of inference rules

- a rich logic library of theories, including types that are particularly suited in the context of computer programming (but not only these)

In this logic it is possible, for instance, to define a type of binary trees and then define a recursive predicate stating when such a tree is a binary search tree (i.e. it is sorted inorder).

The Theory of Lists

The modules in this library contain the definition of lists as an inductive (algebraic) type, together with many related useful definitions. We show below the defined membership predicate `mem` and the *inductive* predicate `sorted`. Inductive predicates allow for inference rule-based definitions, and are free from termination requirements.

```
type list 'a = Nil | Cons 'a (list 'a)

predicate mem (x: 'a) (l: list 'a) =
    match l with
    | Nil      -> false
    | Cons y r -> x = y \/ mem x r
    end

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 (Co
ns y l))

```

Let us create a module to perform some basic reasoning involving lists. This will illustrate the use of proof goals and lemmas , the difference being that goals are not kept in the context, whereas lemmas are kept and may help subsequent goals to be proved.

The two following results involve list membership and sorted lists.

```
module FirstExample
    use list.List
    use list.Mem
```

```
    use list.Append
    use list.HdTlNoOpt
    use list.SortedInt
    use int.Int

    lemma lm1 : forall l :list 'a, x y: 'a.
        mem x l -> mem x (Cons y l)

    goal g1 : forall x :int, l :list int.
        x <= hd l /\ sorted l -> sorted (Cons x l)

end
```

Proofs and External Tools

Unlike SAT and SMT solvers, Why3 works with validity rather than satisfiability.
In particular, proving that a goal follows from a set of axioms is a consequence problem.

Why3 generates a proof obligation for each lemma and goal in a given module, that is then encoded and dispatched to an external proof tool.

Some proof tools, like interactive proof assistants, are based on logic inference mechanisms that allow for validity to be established directly, whereas others, like SMT solvers, are based on the notion of satisfiability.

Why3 will in the latter case encode the goal/lemma as a satisfiability problem, consisting of the axioms of the current theory, together with the negation of the goal.

For instance the `lemma lm1` will be encoded as an SMT file ending with the following lines:

```
( ... )
;; Goal "lm1"
(assert
  (not
  (forall ((l (list a1)) (x3 a1) (y1 a1))
    (=> (mem1 x3 l) (mem1 x3 (Cons y1 l))))))

(check-sat)

```

Recall that a satisfiability solver may return one of three possible outcomes:

- **sat (Satisfiable)**
  There exists at least one assignment (set of values for the variables) that makes the problem true.
  Thus the original formula is invalid : the proof goal is not a consequence of the theory defined by the axioms.

- **unsat (Unsatisfiable)**
  No satisfying assignment exists for the problem (the constraints cannot all be simultaneously satisfied).
  Thus the original formula is valid : the proof goal is a consequence of the theory defined by the axioms.

- **unknown**
  The solver is unable to determine whether the problem is satisfiable or not within the given time/memory bounds
  Thus it is unknown whether the proof goal is a consequence of the theory or not.

Why3 includes a number of automated strategies that will attempt to prove goals by performing simple proof transformations (such as splitting goals into smaller formulas) and sending the resulting proof obligations in turn to several SMT solvers, if more than one is available.

Both the lemma and the goal in the above example can be proved easily. But when some result is identified as **invalid**, we can ask Why3 to display a **counterexample**.

Counterexamples are simply models for the encoded satisfiability problems (a theory together with the negation of a given goal). A model for such a satisfiability problem is a counterexample for the original validity problem.

Reasoning about Functional Programs

The List library also contains definitions of basic program functions, written in the WhyML programming language. We show two examples, the *length* and *append* polymorphic functions:

```
let rec function length (l: list 'a) : int =
    match l with
    | Nil      -> 0
    | Cons _ r -> 1 + length r
    end

let rec function (++) (l1 l2: list 'a) : list 'a =
    match l1 with
    | Nil -> l2
    | Cons x1 r1 -> Cons x1 (r1 ++ l2)
    end

```

Program functions may have specification contracts. Consider for instance the `Quant` module of the list library:

```
module Quant
  use List  use Mem

  let rec function for_all (p: 'a -> bool) (l:list 'a) : bool
    ensures { result <-> forall x. mem x l -> p x }
  = match l with
    | Nil -> true
    | Cons x r -> p x && for_all p r
    end

  let rec function for_some (p: 'a -> bool) (l:list 'a) : bool
    ensures { result <-> exists x. mem x l /\ p x }
  = match l with
    | Nil -> false
    | Cons x r -> p x || for_some p r
    end

  let function mem (eq:'a -> 'a -> bool) (x:'a) (l:list 'a) : bool
    ensures  { result <-> exists y. mem y l /\ eq x y }
  = for_some (eq x) l

end
```

Note that:

- The above three functions are *higher-order,* taking as parameter a predicate on type `'a`.
- Their definitions include a postcondition (**`ensures`**) where the **`result`** keyword refers to the return value of the function.
- Since all 3 functions have return type `bool`, the postconditions are equivalence formulas involving the result (and some quantified formula).
- The library contains two membership predicates!
  - The first (**`predicate`** `mem ...`) is part of the logic language and cannot be called by program functions.
    - It uses the logic equality predicate
  - The second (**`let function`** `mem …`) is a program function
    - It uses an equality predicate passed as argument
    - Its postcondition uses the logic predicate `mem`!

Functional Programming: Insertion Sort

As an example of how to verify the behavior of functional programs with Why3, we will consider the insertion sort algorithm. We will:

1. Write the algorithm in WhyML
2. Write a specification for it using pre- and postconditions
3. Use the web interface or the Why3 IDE to
   a. generate verification conditions,
   b. export them to an SMT solver, and
   c. discharge them, thus proving that the algorithm is correct w.r.t. the specification

We create a module and start by importing the integer and lists library modules, as well as specific modules containing definitions involving list permutations and sorting.

To define the sorting algorithm we start by writing an ordered insertion function `insert`. Its specification states that

- it should receive a sorted list (the `requires` keyword introduces a *precondition*)
- it will also return a sorted list (the `ensures` keyword introduces a *postcondition*)
- moreover, the result contains the same multiset of elements as the input list, extended with the inserted element. In other words, the output list is a permutation of the elements in the input list, together with the inserted element

```
1   let rec function insert (i: int) (l: list int) : list i
nt
2     requires { sorted l }
3     ensures  { sorted result }
4     ensures  { permut result (Cons i l) }
5   =
6     match l with
7     | Nil -> Cons i Nil
8     | Cons h t -> if i <= h then Cons i l
9                   else Cons h (insert i t)
10    end
11
```

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 allow for the function's contract to be established, in a recursive function the contract itself plays the role of invariant: the very contract of the function that is being verified is used to generate a verification condition regarding the recursive call.

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). This means that the function is correct: if you feed it any sorted list and an integer, the function will indeed insert it in the list and return a sorted list.

The same remarks apply to the `iSort` function. Its specification states that the result is a sorted list that is a permutation of the input list, and the algorithm itself is well-know:

```
  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
```

This is equally proved correct using an automatic strategy.

Finally, we note that Why3 also contains an interpreter. We can run `iSort` on an input list by including the following main function in the module:

```
  let main () =
      let l = Cons 2 (Cons 3 (Cons 1 Nil)) in
      iSort l

end
```

This complete program could now be extracted as an OCaml program (this is outside the scope of this course).

TryWhy3

This online version of Why3 runs on the browser.

- It contains a single SMT solver (Alt-Ergo).
- It allows the user to split goals and increase the time limit allowed to the solver for each goal.
- It does not allow for counter-examples to be inspected.

The limitations (in available solvers and proof transformations) means that some developments may need additional lemmas (and/or asserts) that are not required if the standalone tool is used.

For example, the following version of `insert` may well be easier to verify automatically than the previous one.
The `assert` helps establishing that Consing `h` with `insert i t` results in a sorted list.

```
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 let t' =  insert i t in
                          assert { forall x :int. mem x
t' -> h<=x } ;
                          Cons h t'
  end
```

```
```

Exercise 1

Consider the functional quicksort algorithm. It is written in WhyML as follows, using a helper function `partition`.

```
  let rec function partition (x:int) (l:list int) : (l1:list int, l2:list int)
    ensures  { true }
  =
    match l with
    | Nil -> (Nil, Nil)
    | Cons y ys -> let (ll, lr) = partition x ys in
                   if y<=x then (Cons y ll, lr)
                   else (ll, Cons y lr)
  end


  let rec function quicksort (l :list int) : list int
    ensures { true }
    variant { length l }
  =
    match l with
    | Nil -> Nil
    | Cons x t -> let (l1, l2) = partition x t in
                  let l1' = quicksort l1 in
                  let l2' = quicksort l2 in
                  l1'++(Cons x l2')
    end

```

Note the inclusion of a variant in the contract of the `quicksort` function. Why3 can automatically establish the termination of functions defined using basic structural induction (e.g. on the tail of a list). This is not the case above, where `l1` and `l2` are obtained by partitioning the tail of the argument list `l`.

We thus need to include a variant. The length of the list is an adequate measure, since `l1` and `l2` are both guaranteed to be shorter than `l` (since they result from partitioning `t` and not `l`).

1. Complete the contracts that capture the expected behaviour of both functions
2. Prove the correctness of both functions using Why3
3. If required, place appropriate `assert`s in the above definitions to facilitate the automatic proofs.

Exercise 2

Follow the instructions in the following module. Note that:

1. you may need to include some `assert`s if you use TryWhy3
2. one of the functions makes use of the finite set module from the standard library, you may look it up using the above link.

```
module ListFunctions
  use int.Int
  use list.List
  use list.Length
```

```
  use list.NumOcc
  use set.Fset

  (* 1. Using the num_occ function from the list library,
       write a contract for the delete function describing
       its behavior in terms of the (multi-)sets of elements
       in the argument and result lists.
       Then prove the function's correctness w.r.t. it.
   *)
  let rec function delete (x:int) (l:list int) : list int
  = match l with
    | Nil -> Nil
    | Cons h t -> if h=x then delete x t
                  else Cons h (delete x t)
    end


  (* 2. Now suppose that the list l does not contain repeated elements.
       Then the delete function can be optimized: when x is found
       there is no need to continue the list traversal.
       Modify the function's body and contract accordingly, and
       prove the correctness of the new version.
   *)
  let rec function deleteOne (x: int) (l:list int) : list int =



  (* 3. The following function calculates the finite set of elements
       in a list (the 'add' function is the union with a singleton set).
       Write a contract and prove the function's correctness.
   *)
  let rec ghost function elements (l:list int) : fset int
  = match l with
    | Nil -> empty
    | Cons h t -> add h (elements t)
    end



  (* 4. Include in the contract of deleteOne a postcondition relating
       'elements result' and 'elements l'.
   *)

end
```