Lecture at École Jeunes Chercheurs en Programmation (EJCP) 2014

Programming and Proving: Practice with FoCaLiZe Exercices

François Pessaux

17 juin 2014

1 Propositional Logic

The aim of the exercises of this section is to introduce proofs in FoCaLiZe on simple properties, making explicit the structure, the steps of these proofs. Even if all the coming properties if this section can be automatically proved by Zenon, you are invited to play the game of manually split proofs in order to understand the underlying mechanisms of proving. This way, when properties will get harder and Zenon won't anymore be able to fully automatically handle them, you'll know how to split the proofs according to the shape of their goals.

All the theorems to prove in this section are available, as skeletons without proofs that you can complete (file log_prop_skell.fcl). **Source code for this section must be compiled using** focalizec **(without anymore options).**

Q1 Prove the following theorem :

```
theorem implications:all a b :bool, a ->(b ->a).
```

Q2 Prove the following theorem :

```
theorem no_allows_anything:all a b :bool, (~a ->(a ->b)).
```

Q3 Prove the following theorem :

```
theorem equivalence:all a b :bool, (a ->b)<->((~a)\/b).
```

Q4 Prove the following theorem :

```
theorem et_impl_b:all a b c :bool, (a /\((b ->~a)/\( b \/c)/\~c))->b.
```

Q5 Prove the following theorem :

```
all pile je_gagne:bool, ((pile ->je_gagne)/\( ~pile ->je_gagne))->je_gagne
.
```

Q6 State the following proposition as a logical property and prove it.

"If stairs designed to go up allow going up, if stairs designed to go down allow going down, if going up is not going down and inversely, then with stairs to go up one cannot go down and with stairs to go down one cannot go up."

2 Proofs on Toplevel Programs

The aim of the exercises of this section is to introduce algorithmic definitions and prove properties on them. These programs do not use FoCaLiZe encapsulation constructs, hence look like regular ML programs.

2.1 Signals

The program of this section aims at representing a simple traffic signal and provide a function telling if, according to the signal's color, one is allowed to pass or not.

A skeleton for this exercise is available in the file (file `signal_skel.fcl`). **Source code for this section must be compiled using `focalizec` (without anymore options).**

```
open "basics" ;;

type signal_t =
 | Red
 | Orange
 | Green ;;

let can_pass (x) =
  match x with
   | Red -> false
   | Orange -> false
   | Green -> true ;;
```

Q7 Prove the theorem stating that *"If one can pass, then the signal is green"*.

Q8 Prove the theorem stating that *"One can pass if and only if the signal is green"*.

2.2 Intervals

The program of this section aims at representing intervals of integers using a structure of pair carrying the invariant that its first component is always the lowest bound of the interval. A function `make` is provided to build an interval complying with the invariant from 2 integers given as argument. Two functions, `first` and `second` are provided to access the lowest (resp. the greatest) bound of an interval given as argument.

A skeleton for this exercise is available in the file (file `interv_skel.fcl`). **Source code for this section must be compiled using `focalizec` (without anymore options).**

```
open "basics" ;;

let make (x, y) : interv_t = if x <=0x y then (x, y) else (y, x) ;;

let first (i: interv_t) = match i with | (x, _) -> x ;;
let second (i: interv_t) = match i with | (_, y) -> y ;;
```

Q9 Prove a theorem stating that the function `make` complies with the invariant previously described.

To deal with $<$, $\leq$, $>$, $\geq$ and friends on integers, the FoCaLiZe standard library provides (in `basics`) the useful theorems (proved directly in `Coq`) you will probably need :

```
theorem int_not_le_gt :  all x y : int, ~ (x <=0x y) -> (x >0x y)
theorem int_gt_implies_ge : all x y : int, x >0x y -> x >=0x y
theorem int_ge_le_swap : all x y : int , (x >=0x y) -> (y <=0x x)
```

A new function `add` is provided to add two intervals.

```
let add (i1 : interv_t, i2: interv_t) =
  let min1 = first (i1) in
  let max1 = second (i1) in
  let min2 = first (i2) in
  let max2 = second (i2) in
  (min1 + min2, max1 + max2) ;;
```

Q10 Prove theorem stating that if an interval `i1` is built calling `make` with 2 and 3, if an interval `i2` is built calling `make` with 5 and 2, then `i3` being the addition of `i1` and `i2` also comply with the invariant.

Again, the `FoCaLiZe` standard library provides (in `basics`) the useful theorem (proved directly in `Coq`) you will probably need :

```
theorem int_le_plus_plus:
  all x y z t : int, x <=0x y -> z <=0x t -> x + z <=0x y + t
```

2.3 Association Maps

The questions of this section address some properties demonstrated in the lecture. It show that using the construct of species of `FoCaLiZe` does not make proofs different nor more complex.

A skeleton for this exercise is available in the file (file `assoc_map1_skel.fcl`). **Source code for this section must be compiled using `focalizec -zvtovopt -script` (with these explicit options).**

```
open "basics" ;;

(* Structure recording bindings of a map: a hand-made basic list. *)
type int_str_list_t =
 | Nil
 | Node (int, string , int_str_list_t)   (* Keys: ints, values: strings. *)
;;

(* Return value of the lookup function: nothing or something. *)
type option_t ('a) =
 | None
 | Some ('a)
;;

species AssocMap =
  representation = int_str_list_t ;
  let empty : Self = Nil ;    (* Empty association map: no bindings. *)

  (* Addition to the map m of the value v bound to the key k. *)
  let add (k: int, v: string, m : Self) : Self = Node (k, v, m) ;

  (* Lookup the the value bound to the key k in the map m. *)
  let rec find (k: int, m: Self) =
    match m with
    | Nil -> None
    | Node (kcur, v, q) -> if kcur = k then Some (v) else find (k, q)
  termination proof = structural m ;
end ;;
```

Q11 Prove the following theorem :

```
(* Add make find a success. *)
theorem find_added_not_fails: all k : int, all v : string, all m1 m2 : Self,
  m2 = add (k, v, m1) -> ~ (find (k, m2) = None)
```

Q12 Prove the following theorem :

```
(* Same key -> same value. *)
theorem find_same_key_same_value: all k1 k2: int, all m : Self,
  k1 = k2 -> find (k1, m) = find (k2, m)
```

Q13 Prove the following theorem :

```
theorem find_spec: all m : Self, all s k : int, all v : string,
  (find (s, m) = Some (v) \/ s = k) <-> find (s, add (k, v, m)) = Some (v)
```

3 Induction

3.1 Again Propositional Calculus

The exercices of this section is to address a more complex proof by induction and to show that encoding propositional calculus through an evaluation does not make proofs of "usual" propositional logic harder.

The following program defines the type of propositional calculus formula and a `simpl` function replacing, in a formula, all equivalences by bi-directional implications.

All the theorems to prove in this section are available, as skeletons without proofs that you can complete (file `case_induct_skell.fcl`). **Source code for this section must be compiled using `focalizec -zvtovopt -script` (with these explicit options).**

```
open "basics" ;;

type logexp_t =
  | T
  | F
  | Not (logexp_t)
  | And (logexp_t, logexp_t)
  | Or (logexp_t, logexp_t)
  | Equiv (logexp_t, logexp_t)
  | Imply (logexp_t, logexp_t) ;;

let rec simpl (e) =
  match e with
   | T -> T
   | F -> F
   | Not (ee) -> Not (simpl (ee))
   | And (ee1, ee2) -> And (simpl (ee1), simpl (ee2))
   | Or (ee1, ee2) -> Or (simpl (ee1), simpl (ee2))
   | Equiv (ee1, ee2) ->
       let see1 = simpl (ee1) in
       let see2 = simpl (ee2) in
       And (Imply (see1, see2), Imply (see2, see1))
   | Imply (ee1, ee2) -> Imply (simpl (ee1), simpl (ee2))
termination proof = structural e ;;
```

Q14 State the theorem telling that after "simplification", a formula does not contain anymore "equivalences". What do you need ?

Q15 Prove the theorem above stated in Q14.

We now provide an evaluation function `eval` returning the boolean value in which a closed formula evaluates.

```
let rec eval (e) =
  match e with
    | T -> true
    | F -> false
    | Not (ee) -> ~~ eval (ee)
    | And (ee1, ee2) -> eval (ee1) && eval (ee2)
    | Or (ee1, ee2) -> eval (ee1) || eval (ee2)
    | Equiv (ee1, ee2) ->
       (~~ eval (ee1) || eval (ee2)) && (~~ eval (ee2) || eval (ee1))
    | Imply (ee1, ee2) -> ~~ eval (ee1) || eval (ee2) ;;
```

Q16 Prove the well-known theorem stating that "equivalence is equivalent to implications in both ways".

Q17 Prove the well-known theorem stating that "a logical formula in propositional calculus evaluates into true or false".