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).`

Solution :

```
theorem implications:  all a b : bool, a -> (b -> a)
proof =
  <1>1 assume a : bool, b : bool,
       hypothesis h1 : a,
       prove b -> a
       <2>1 hypothesis h2 : b,
            prove a
            by hypothesis h1
       <2>2 qed by step <2>1
  <1>2 conclude ;;
```

Q2 Prove the following theorem :
`theorem no_allows_anything:all a b :bool, (~a ->(a ->b)).`

Solution :

```
theorem no_allows_anything: all a b : bool, (~ a -> (a -> b))
proof =
  <1>1 assume a : bool, b : bool,
       prove ~ a -> ( a -> b)
       <2>1 hypothesis h1: ~ a,
            prove  a -> b
            <3>1 hypothesis h2 : a,
```

```
                prove b
                by hypothesis h1, h2
             <3>e qed by step <3>1
        <2>e qed by step <2>1
    <1>e conclude ;;
```

Q3 Prove the following theorem :
theorem equivalence:**all** a b :bool, (a **->**b)<**->**((~a)**\/**b).

Solution :

```
theorem equivalence: all a b : bool, (a -> b) <-> ((~ a) \/ b)
proof =
  <1>1 assume a : bool, b : bool,
       prove  (a -> b) <-> (~ a \/ b)
       <2>1 prove  (a -> b) -> (~ a \/ b)
            <3>1 hypothesis h1 : a -> b,
                 prove ~ a \/ b
                 <4>1 hypothesis h2: a,
                      prove b by hypothesis h1, h2
                 <4>2 hypothesis h3: ~ b,
                      prove ~ a by hypothesis h1, h3
                 <4>e qed by step <4>1, <4>2
            <3>e qed by step <3>1
       <2>2 prove (~ a \/ b) -> (a -> b)
            <3>1 hypothesis h4 : ~ a \/ b,
                 prove a -> b
                 <4>1 hypothesis h5: a,
                       prove b by hypothesis h4, h5
                 <4>e qed conclude
            <3>e qed by step <3>1
       <2>e qed by step <2>1, <2>2
  <1>e conclude ;;
```

Q4 Prove the following theorem :
theorem et_impl_b:**all** a b c :bool, (a **/**((b **->**~a)**/**(b **\/**c)**/**~c))**->**b.

Solution :

```
proof =
  <1>1 assume a : bool, b : bool, c : bool,
       prove (a /\ ((b -> ~ a) /\ ( b \/ c) /\ ~ c)) -> b
       <2>1 hypothesis h1: a /\ ((b -> ~ a) /\ ( b \/ c) /\ ~ c),
            prove b
            <3>1 hypothesis h2: ~ b, (* Absurdum. *)
                 prove false
                 <4>1 prove c by hypothesis h1, h2
                 <4>2 prove ~ c by hypothesis h1, h2
                 <4>e qed by step <4>1, <4>2
            <4>1 qed by step <3>1
       <2>e qed by step <2>1
  <1>e conclude ;;
```

Q5 Prove the following theorem :
all pile je_gagne:bool, ((pile **->**je_gagne)**/**(~pile **->**je_gagne))**->**je_gagne
.

Solution :

```
theorem pile_ou_face :
  all pile je_gagne : bool,
    ((pile -> je_gagne) /\ ( ~ pile -> je_gagne)) -> je_gagne
proof =
  <1>1 assume pile : bool, je_gagne : bool,
       prove  ((pile -> je_gagne) /\ ( ~ pile -> je_gagne)) -> je_gagne
       <2>1 hypothesis h1 : (pile -> je_gagne) /\ ( ~ pile -> je_gagne),
            prove je_gagne
            <3>1 hypothesis h2: pile,
                 prove je_gagne by hypothesis h1, h2
            <3>2 hypothesis h3: ~ pile,
                 prove je_gagne by hypothesis h1, h3
            <3>e qed by step <3>1, <3>2
       <2>e qed by step <2>1
  <1>e conclude ;;
```

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."

Solution :

```
theorem shadock:
  all escalier_pour_monter: bool, all monte: bool,
  all escalier_pour_descendre: bool, all descend : bool,
    (escalier_pour_monter -> monte) ->
    (escalier_pour_descendre -> descend) ->
    (monte <-> ~ descend) ->
    ((escalier_pour_monter -> ~ descend) /\
     (escalier_pour_descendre -> ~ monte))
proof =
  <1>1 assume escalier_pour_monter monte : bool,
       assume escalier_pour_descendre descend : bool,
       prove (escalier_pour_monter -> monte) ->
         (escalier_pour_descendre -> descend) ->
         (monte <-> ~ descend) ->
         ((escalier_pour_monter -> ~ descend) /\
          (escalier_pour_descendre -> ~ monte))
       <2>1 hypothesis h1: escalier_pour_monter -> monte,
            hypothesis h2: escalier_pour_descendre -> descend,
            hypothesis h3: monte <-> ~ descend,
            prove (escalier_pour_monter -> ~ descend) /\
                  (escalier_pour_descendre -> ~ monte)
            <3>1 prove escalier_pour_monter -> ~ descend
                 by hypothesis h1, h3
            <3>2 prove escalier_pour_descendre -> ~ monte
                 by hypothesis h2, h3
            <3>e qed by step <3>1, <3>2
       <2>e conclude
  <1>e conclude ;;
```

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"*.

Solution :

```
theorem pass_at_green: all s : signal_t, (can_pass (s) = true) -> s = Green
proof =
  <1>1 assume s : signal_t,
       hypothesis H: can_pass (s) = true,
       prove s = Green by definition of can_pass type signal_t hypothesis H
  <1>e qed by step <1>1 ;;
```

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

Solution :

```
theorem pass_only_at_green:
  all s : signal_t, (can_pass (s) = true) <-> s = Green
proof =
  <1>1 prove all s: signal_t, (can_pass (s) = true) -> s = Green
       by property pass_at_green
  <1>2 prove all s: signal_t,  s = Green -> (can_pass (s) = true)
       <2>1 assume s : signal_t,
            hypothesis H: s = Green,
            prove can_pass (s) = true
            by definition of can_pass type signal_t hypothesis H
       <2>e conclude
  <1>e conclude ;;
```

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)
```

Solution :

```
theorem make_invariant: all x y : int, all i : interv_t,
  i = make (x, y) -> first (i) <=0x second (i)
proof =
  <1>1 assume x y : int,
       assume i : interv_t,
       hypothesis h1: i = make (x, y),
       prove first (i) <=0x second (i)
       <2>1 hypothesis h2: x <=0x y,
            prove first (i) <=0x second (i)
            <3>1 prove first (i) = x
                 by definition of make, first hypothesis h1, h2
            <3>2 prove second (i) = y
                 by definition of make, second hypothesis h1, h2
            <3>e qed by step <3>1, <3>2 hypothesis h2
       <2>2 hypothesis h3: ~ x <=0x y,
            prove first (i) <=0x second (i)
            <3>1 prove y <=0x x
                 <4>1 prove x >0x y by hypothesis h3 property int_not_le_gt
                 <4>2 prove x >=0x y by step <4>1 property int_gt_implies_ge
                 <4>e qed by step <4>2 property int_ge_le_swap
            <3>2 prove first (i) = y
                 by definition of first, make step <3>1 hypothesis h1, h3
            <3>3 prove second (i) = x
                 by definition of second, make step <3>1 hypothesis h1, h3
            <3>e qed by step <3>1, <3>2, <3>3
       <2>e qed by step <2>1, <2>2
  <1>e conclude ;;
```

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
```

Solution :

```
theorem add_invariant: all i1 i2 i3: interv_t,
  i1 = make (2, 3) -> i2 = make (5,2) -> i3 = add (i1, i2) ->
    first (i3) <=0x second (i3)
proof =
  <1>1 assume i1 i2 i3: interv_t,
       hypothesis h1: i1 = make (2, 3),
       hypothesis h2: i2 = make (5,2),
       hypothesis h3: i3 = add (i1, i2),
       prove first (i3) <=0x second (i3)
       <2>1 prove first (i3) = first (i1) + first (i2)
            by definition of add, first hypothesis h3
       <2>2 prove second (i3) = second (i1) + second (i2)
            by definition of add, second hypothesis h3
       <2>3 prove (first (i1) + first (i2)) <=0x (second (i1) + second (i2))
            <3>1 prove first (i1) <=0x second (i1) assumed
            <3>2 prove first (i2) <=0x second (i2) assumed
            <3>e qed by step <3>1, <3>2 property int_le_plus_plus
       <2>e qed by step <2>1, <2>2, <2>3
  <1>e conclude ;;
```

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)
```

Solution :

```
theorem find_added_not_fails: all k : int, all v : string, all m1 m2 : Self,
  m2 = add (k, v, m1) -> ~ (find (k, m2) = None)
proof = by definition of add, find type int_str_list_t, option_t ;
```

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)
```

Solution :

```
theorem find_same_key_same_value: all k1 k2: int, all m : Self,
  k1 = k2 -> find (k1, m) = find (k2, m)
proof = conclude ;
```

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)
```

Solution :

```
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)
proof =
  <1>1 assume m : Self,
       assume s k : int,
       assume v : string,
       prove (find (s, m) = Some (v) \/ s = k) <->
             find (s, add (k, v, m)) = Some (v)
       <2>1 hypothesis H1: find (s, m) = Some (v) \/ s = k,
            prove find (s, add (k, v, m)) = Some (v)
            <3>1 prove add (k, v, m) = Node (k, v, m)
                 by definition of add type int_str_list_t
            <3>e qed by step <3>1 definition of find
                 hypothesis H1 type int_str_list_t
       <2>2 hypothesis H2: find (s, add (k, v, m)) = Some (v),
            prove find (s, m) = Some (v) \/ s = k
            <3>1 prove add (k, v, m) = Node (k, v, m)
                 by definition of add type int_str_list_t
            <3>e qed by step <3>1 definition of find
                 hypothesis H2 type int_str_list_t
       <2>e qed by step <2>1, <2>2
  <1>e conclude ;
```

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 ?

Solution : We need a function `contains_equiv` checking if a formula contains equivalence nodes. In effect, the "contain" property cannot be expressed as pure logical connectors combination. It requires to perform a computation. . . exactly what does an algorithm, a function. Especially, "contains" must not be confused with $\exists$.

```
let rec contains_equiv (e) =
  match e with
   | T -> false
   | F -> false
   | Not (ee) -> contains_equiv (ee)
   | And (ee1, ee2) -> contains_equiv (ee1) || contains_equiv (ee2)
   | Or (ee1, ee2) -> contains_equiv (ee1) || contains_equiv (ee2)
   | Equiv (_ee1, _ee2) -> true
   | Imply (ee1, ee2) -> contains_equiv (ee1) || contains_equiv (ee2)
termination proof = structural e ;;

theorem no_more_equiv_after_simpl: all e: logexp_t, ~ contains_equiv (simpl (e))
```

Q15 Prove the theorem above stated in Q14.

Solution :

```
theorem no_more_equiv_after_simpl: all e: logexp_t, ~ contains_equiv (simpl (e))
proof =
  <1>1 prove ~ contains_equiv (simpl (T))
       by definition of contains_equiv, simpl type logexp_t
  <1>2 prove ~ contains_equiv (simpl (F))
       by definition of contains_equiv, simpl type logexp_t
  <1>3 assume e1: logexp_t,
       hypothesis hrec: ~ contains_equiv (simpl (e1)),
```

```
      prove ~ contains_equiv (simpl (Not (e1)))
      (* Let's just prove that simpl on the Not is simpl on the argument,
         hence, this doesn't change the result of contains_equiv. *)
      <2>1 prove simpl (Not (e1)) = Not (simpl (e1))
           by definition of simpl type logexp_t
      <2>e qed by step <2>1 hypothesis hrec definition of contains_equiv
           type logexp_t
  <1>4 assume e1: logexp_t,
      hypothesis hrec1: ~ contains_equiv (simpl (e1)),
      assume e2: logexp_t,
      hypothesis hrec2: ~ contains_equiv (simpl (e2)),
      prove ~ contains_equiv (simpl (And (e1, e2)))
      (* Same idea for all similar cases. *)
      <2>1 prove simpl (And (e1, e2)) = And (simpl (e1), simpl (e2))
           by definition of simpl type logexp_t
      <2>e qed by step <2>1 hypothesis hrec1, hrec2
           definition of contains_equiv type logexp_t
  <1>5 assume e1: logexp_t,
      hypothesis hrec1: ~ contains_equiv (simpl (e1)),
      assume e2: logexp_t,
      hypothesis hrec2: ~ contains_equiv (simpl (e2)),
      prove ~ contains_equiv (simpl (Or (e1, e2)))
      (* Same idea for all similar cases. *)
      <2>1 prove simpl (Or (e1, e2)) = Or (simpl (e1), simpl (e2))
           by definition of simpl type logexp_t
      <2>e qed by step <2>1 hypothesis hrec1, hrec2
           definition of contains_equiv type logexp_t
  <1>6 assume e1: logexp_t,
      hypothesis hrec1: ~ contains_equiv (simpl (e1)),
      assume e2: logexp_t,
      hypothesis hrec2: ~ contains_equiv (simpl (e2)),
      prove ~ contains_equiv (simpl (Equiv (e1, e2)))
      <2>1 prove simpl (Equiv (e1, e2)) =
                  And (Imply (simpl (e1), simpl (e2)),
                       Imply (simpl (e2), simpl (e1)))
           by definition of simpl type logexp_t
      <2>e qed by step <2>1 hypothesis hrec1, hrec2 type logexp_t
           definition of contains_equiv
  <1>7 assume e1: logexp_t,
      hypothesis hrec1: ~ contains_equiv (simpl (e1)),
      assume e2: logexp_t,
      hypothesis hrec2: ~ contains_equiv (simpl (e2)),
      prove ~ contains_equiv (simpl (Imply (e1, e2)))
      (* Same idea for all similar cases. *)
      <2>1 prove simpl (Imply (e1, e2)) = Imply (simpl (e1), simpl (e2))
           by definition of simpl type logexp_t
      <2>e qed by step <2>1 hypothesis hrec1, hrec2
           definition of contains_equiv type logexp_t
  <1>e qed by step <1>1, <1>2, <1>3, <1>4, <1>5, <1>6, <1>7 type logexp_t ;;
```

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".

Solution :

```
theorem equiv_is_twice_imply: all e1 e2 : logexp_t,
   eval (Equiv (e1, e2)) <-> (eval (Imply (e1, e2)) /\ eval (Imply (e2, e1)))
proof = by type logexp_t definition of eval ;;
```

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

Solution :

```
theorem eval_returns_true_or_false: all e : logexp_t, eval (e) \/ ~ eval (e)
proof = by type logexp_t definition of eval ;;
```