Infrastructure de Tests - Rapport

Hamza Mouddene
Younes Saoudi
Oussama Echcherqaoui
Manal Hajji

January 24, 2021

Contents

1 Introduction

Ce projet s'intéresse à la programmation non-déterministe qui permet, à l'aide d'opérations non standard, de considérer plusieurs exécutions d'un même programme, d'en filtrer/éliminer certaines, de quantifier (universel ou existentiel) sur les exécutions possibles, etc. On implantera plus particulièrement des fonctionnalités qui permettent simplement la mise en place d'une infrastructure de tests généralisée, avec des contrats de fonction, des fonctions partiellement ou pas implantées, etc. Dans ce cadre, la principale caractéristique associée à une exécution est qu'elle peut être seulement valide ou invalide et ne possède pas de valeur calculée. Une exécution ordinaire d'un programme ordinaire est valide par défaut, i.e. on ignore son résultat et on suppose que les programmes manipulés ne provoquent pas d'erreur ou d'exception.

2 Architecture

Le projet se compose de cinq fichiers dont deux se sont les fichiers de configuration de dune, les trois restants sont *main.mli*, *main.ml* et *tests.ml*.

- *main.mli* : c'est une interface qui contient la déclaration et a spécification du flux ainsi que les fonctions minimum.
- *main.ml* : c'est un module conforme à l'interface précédente.
- *test.ml* : C'est les tests fournis dans le sujet du projet.

Figure 1: L'architecture du projet

3 Conception

3.1 Fonctionnalités minimum

- assumption : (unit → bool) → unit, qui filtre et ne continue que les exécutions qui vérifient le prédicat passé en paramètre. Les autres exécutions sont simplement arrêtées et déclarées valides.

```
(*
       Filters and continues just the executions that check the given predicate in
       parameter. The other executions should be stopped and declared valid.
       Signature     : assumption : (unit -> bool) -> unit = <fun>.
       Parameter(s)  : The predicate to check.
       Precondtion   : Check if the predicate is valid.
       Postcondtion  : None.
       Result        : None.
*)
let assumption f = match f() with
       | false -> let _ = miracle in ()
       | true ->  continue();;
```

Figure 2: La fonction assumption

- assertion : (unit → bool) → unit, qui filtre et ne continue que les exécutions qui vérifient le prédicat passé en paramètre. Les autres exécutions sont simplement arrêtées et déclarées invalides.

```
(*
       Filters and continues just the executions that check the given predicate in
       parameter. The other executions should be stopped and declared invalid.
       Signature     : assertion : (unit -> bool) -> unit = <fun>.
       Parameter(s)  : The predicate to check.
       Precondtion   : None.
       Postcondtion  : Check if the predicate is valid.
       Result        : None.
*)
let assertion f =  match f() with
       | false -> let _ = failure in ()
       | true ->  continue();;
```

Figure 3: La fonction assertion

- miracle : unit → 'a, interrompt l'exécution et la rend valide.

```
(*
        Interrupt the execution and makes it valid.
        Signature     : miracle : unit -> 'a = <fun>.
        Parameter(s)  : None.
        Precondtion   : None.
        Postcondtion  : None.
        Result        : None.
*)
let miracle () = Delimcc.shift prompt0 (fun x -> x Valid);;
```

Figure 4: La fonction assertion

- failure : unit → 'a, interrompt l'exécution et la rend invalide.

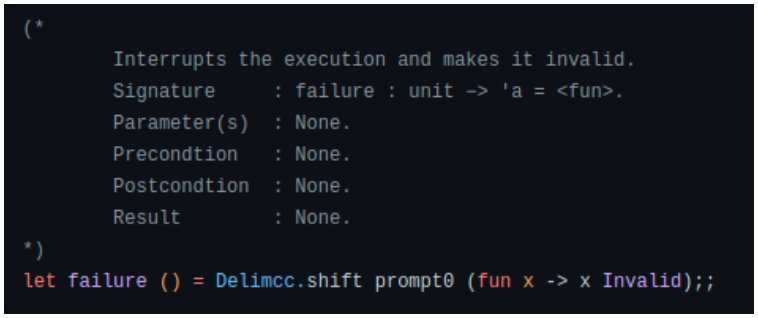


Figure 5: La fonction assertion

- forall bool : unit → bool, qui "forke" l'exécution courante en deux versions. Dans chacune de ces versions, forall bool renvoie un booléen différent. L'exécution "parente" est valide si et seulement si les deux exécutions "filles" le sont.

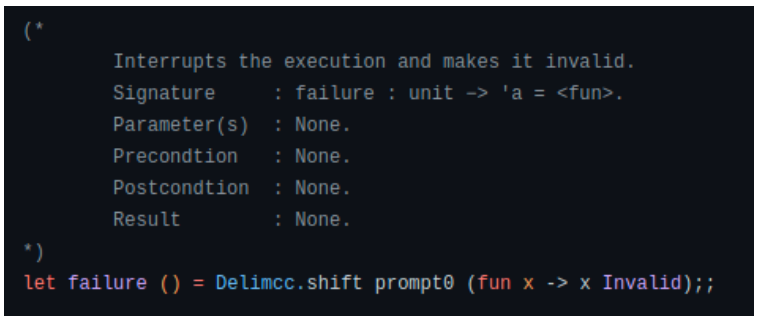


Figure 6: La fonction forall_bool

- forsome bool : unit → bool, qui "forke" de la même façon l'exécution courante. L'exécution "parente" est valide si et seulement si au moins une exécution "fille" l'est.

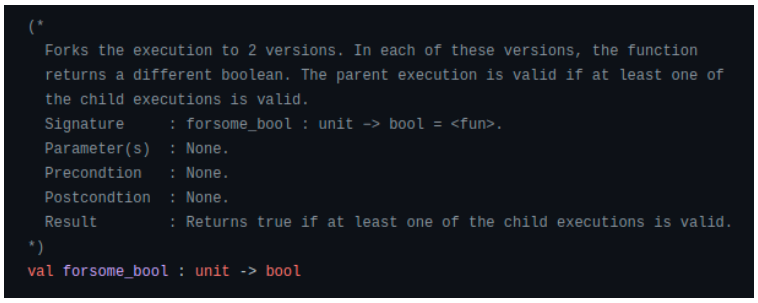
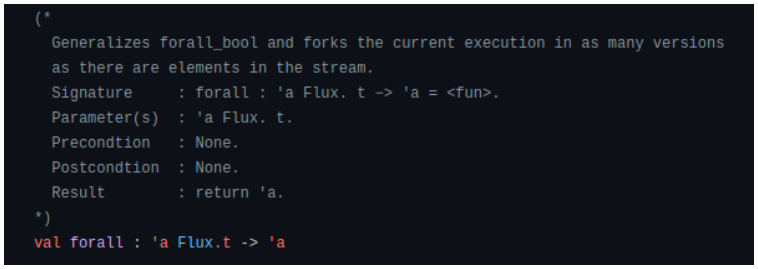


Figure 7: La fonction forsome_bool

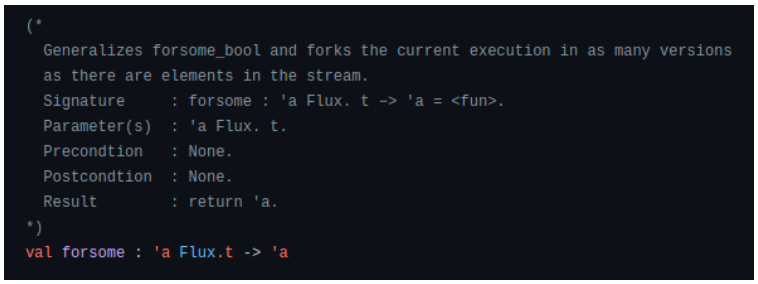
- forall : 'a Flux. t → 'a, qui généralise forall bool et "fork" l'exécution courante en autant de versions qu'il y a d'éléments dans le flux.



```
(*
  Generalizes forall_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature     : forall : 'a Flux. t -> 'a = <fun>.
  Parameter(s)  : 'a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : return 'a.
*)
val forall : 'a Flux.t -> 'a
```

Figure 8: La fonction forall

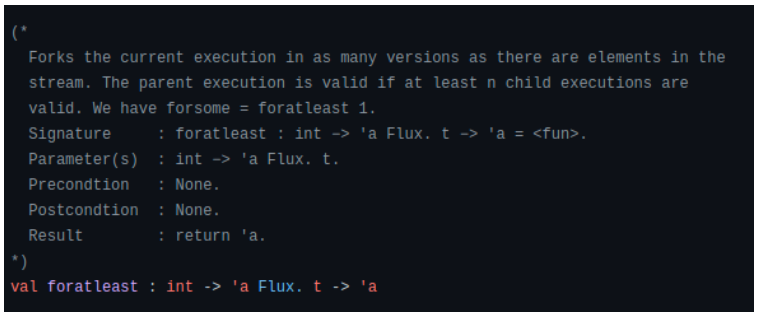
- forsome : 'a Flux. t → 'a, qui "forke" de la même façon l'exécution courante et généralise forsome bool.



```
(*
  Generalizes forsome_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature     : forsome : 'a Flux. t -> 'a = <fun>.
  Parameter(s)  : 'a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : return 'a.
*)
val forsome : 'a Flux.t -> 'a
```

Figure 9: La fonction forsome

- foratleast : int → 'a Flux.t → 'a, qui "forke" de la même façon l'exécution courante. L'exécution "parente" est valide si et seulement si au moins n exécutions "filles" le sont. On à forsome = foratleast 1.



```
(*
  Forks the current execution in as many versions as there are elements in the
  stream. The parent execution is valid if at least n child executions are
  valid. We have forsome = foratleast 1.
  Signature     : foratleast : int -> 'a Flux. t -> 'a = <fun>.
  Parameter(s)  : int -> 'a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : return 'a.
*)
val foratleast : int -> 'a Flux. t -> 'a
```

Figure 10: La fonction foratleast

- check : (unit → unit) → bool, qui exécute un programme instrumenté avec les primitives ci-dessus. Le résultat booléen représente la validité de l'exécution et permet de s'interfacer avec **let** (%test de) ppx_inline_test.

```
(*
  Executes a program containing the functions above.
  Signature     : forsome : 'a Flux. t -> 'a = <fun>.
  Parameter(s)  : None.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : Return true if the execution is valid, otherwise false.
*)
val check : ( unit -> unit) -> bool
```

Figure 11: La fonction check

3.2 Extension

Nous avons implanté la première extension contenant les fonctions:

```
(* Extensions *)
val forall_length: int Flux.t -> (unit -> 'a) -> 'a list
val forsome_length : int Flux.t -> (unit -> 'a) -> 'a list
val foratleast_length : int -> int Flux.t -> (unit -> 'a) -> 'a list
```

Figure 12: Les Fonctions de l'Extension Implantée

3.3 Test

Le fichier tests.ml (voir **Annexe**) contient une dizaine de tests de l'infrastructure avec son extension.

4 Conclusion

Pour conclure, ce projet nous a aidé à évoluer dans le cadre de la programmation fonctionnelle. Nous avons acquis de nouvelles compétences même si nous n'avons pas pu tout implanter correctement.

5 Annexe

5.1 Infrastructure de tests

```
(* interfaces des flux utiles pour toute la séance *)

module type Iter =
sig
  type 'a t
  val vide : 'a t
  val cons : 'a -> 'a t -> 'a t
  val uncons : 'a t -> ('a * 'a t) option
  val unfold : ('s -> ('a * 's) option) -> 's -> 'a t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val constant : 'a -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val apply : ('a -> 'b) t -> 'a t -> 'b t
  val isEmpty : 'a t -> bool
end


(* Module Flux implantant l'interface de flux Iter *)
(* a l'aide d'une structure de donnees paresseuse  *)
type 'a flux = Tick of ('a * 'a flux) option Lazy.t;;
module Flux : Iter with type 'a t = 'a flux;;
type result;;

  (*
    Filters and continues just the executions that check the given predicate in
    parameter. The other executions should be stopped and declared valid.
    Signature      : assumption : (unit > bool) > unit = <fun>.
    Parameter(s)   : The predicate to check.
    Precondtion    : Check if the predicate is valid.
    Postcondtion   : None.
    Result         : None.
  *)
  val assumption : (unit -> bool) -> unit

  (*
    Filters and continues just the executions that check the given predicate in
    parameter. The other executions should be stopped and declared invalid.
    Signature      : assertion : (unit > bool) > unit = <fun>.
    Parameter(s)   : The predicate to check.
    Precondtion    : None.
    Postcondtion   : Check if the predicate is valid.
    Result         : None.
  *)
  val assertion : (unit -> bool) -> unit

  (*
    Interrupt the execution and makes it valid.
    Signature      : miracle : unit > 'a = <fun>.
    Parameter(s)   : None.
    Precondtion    : None.
    Postcondtion   : None.
    Result         : None.
  *)
  val miracle : unit -> result

  (*
    Interrupts the execution and makes it invalid.
```

```
  Signature      : failure : unit > 'a = <fun>.
  Parameter(s)   : None.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : None.
*)
val failure : unit -> result

(*
  Forks the execution to 2 versions. In each of these versions, the function
  returns a different boolean. The parent execution is valid if only the child
  executions are valid.
  Signature      : forall_bool : unit > bool = <fun>.
  Parameter(s)   : None.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : Return true if the child executions are valid.
*)
val forall_bool : unit -> bool

(*
  Forks the execution to 2 versions. In each of these versions, the function
  returns a different boolean. The parent execution is valid if at least one of
  the child executions is valid.
  Signature      : forsome_bool : unit > bool = <fun>.
  Parameter(s)   : None.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : Returns true if at least one of the child executions is valid.
*)
val forsome_bool : unit -> bool

(*
  Generalizes forall_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature      : forall : 'a Flux. t > 'a = <fun>.
  Parameter(s)   : 'a Flux. t.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : return 'a.
*)
val forall : 'a Flux.t -> 'a

(*
  Generalizes forsome_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature      : forsome : 'a Flux. t > 'a = <fun>.
  Parameter(s)   : 'a Flux. t.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : return 'a.
*)
val forsome : 'a Flux.t -> 'a

(*
  Forks the current execution in as many versions as there are elements in the
  stream. The parent execution is valid if at least n child executions are
  valid. We have forsome = foratleast 1.
  Signature      : foratleast : int > 'a Flux. t > 'a = <fun>.
  Parameter(s)   : int > 'a Flux. t.
  Precondtion    : None.
  Postcondtion   : None.
  Result         : return 'a.
*)
val foratleast : int -> 'a Flux. t -> 'a
```

```
125
126 (*
127   Executes a program containing the functions above.
128   Signature     : forsome : 'a Flux. t > 'a = <fun>.
129   Parameter(s)  : None.
130   Precondtion   : None.
131   Postcondtion  : None.
132   Result        : Return true if the execution is valid, otherwise false.
133 *)
134 val check : ( unit -> unit) -> bool
135
136 (* Extensions *)
137 val forall_length: int Flux.t -> (unit -> 'a) -> 'a list
138 val forsome_length : int Flux.t -> (unit -> 'a) -> 'a list
139 val foratleast_length : int -> int Flux.t -> (unit -> 'a) -> 'a list
```

```
module type Iter =
sig
  type 'a t
  val vide : 'a t
  val cons : 'a -> 'a t -> 'a t
  val uncons : 'a t -> ('a * 'a t) option
  val unfold : ('s -> ('a * 's) option) -> 's -> 'a t
  val filter : ('a -> bool) -> 'a t -> 'a t
  val append : 'a t -> 'a t -> 'a t
  val constant : 'a -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val apply : ('a -> 'b) t -> 'a t -> 'b t
  val isEmpty : 'a t -> bool
end

type 'a flux = Tick of ('a * 'a flux) option Lazy.t;;

module Flux : Iter with type 'a t = 'a flux =
  struct
    type 'a t = 'a flux = Tick of ('a * 'a t) option Lazy.t;;

    let vide = Tick (lazy None);;

    let cons t q = Tick (lazy (Some (t, q)));;

    let uncons (Tick flux) = Lazy.force flux;;

    let rec apply f x =
      Tick (lazy (
      match uncons f, uncons x with
      | None         , _             -> None
      | _            , None          -> None
      | Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx)));;

    let rec unfold f e =
      Tick (lazy (
      match f e with
      | None         -> None
      | Some (t, e') -> Some (t, unfold f e')));;

    let rec filter p flux =
      Tick (lazy (
      match uncons flux with
      | None        -> None
      | Some (t, q) -> if p t then Some (t, filter p q)
                       else uncons (filter p q)));;

    let rec append flux1 flux2 =
      Tick (lazy (
      match uncons flux1 with
      | None          -> uncons flux2
      | Some (t1, q1) -> Some (t1, append q1 flux2)));;

    let constant c = unfold (fun () -> Some (c, ())) ();;
    (* implantation rapide mais inefficace de map *)
    let map f i = apply (constant f) i;;

    let map2 f i1 i2 = apply (apply (constant f) i1) i2;;

    let isEmpty stream =
      match (uncons stream) with
        | None -> true
        | _ -> false;;
  end
```

```
66
67
68
69 let prompt0 = Delimcc.new_prompt();;
70 type result = | Invalid | Valid ;;
71
72 (*
73   Interrupt the execution and makes it valid.
74   Signature      : miracle : unit > 'a = <fun>.
75   Parameter(s)   : None.
76   Precondtion    : None.
77   Postcondtion   : None.
78   Result         : None.
79 *)
80 let miracle () = Delimcc.shift prompt0 (fun x -> x Valid);;
81
82 (*
83   Interrupts the execution and makes it invalid.
84   Signature      : failure : unit > 'a = <fun>.
85   Parameter(s)   : None.
86   Precondtion    : None.
87   Postcondtion   : None.
88   Result         : None.
89 *)
90 let failure () = Delimcc.shift prompt0 (fun x -> x Invalid);;
91
92 (*
93   Continues the execution
94   Signature      : continue : unit > 'a = <fun>.
95   Parameter(s)   : None.
96   Precondtion    : None.
97   Postcondtion   : None.
98   Result         : None.
99 *)
100 let continue () = Delimcc.shift prompt0 (fun x -> x ());;
101
102 (*
103   Filters and continues just the executions that check the given predicate in
104   parameter. The other executions should be stopped and declared valid.
105   Signature      : assumption : (unit > bool) > unit = <fun>.
106   Parameter(s)   : The predicate to check.
107   Precondtion    : Check if the predicate is valid.
108   Postcondtion   : None.
109   Result         : None.
110 *)
111 let assumption f = match f() with
112   | false -> let _ = miracle in ()
113   | true ->  continue();;
114
115
116 (*
117   Filters and continues just the executions that check the given predicate in
118   parameter. The other executions should be stopped and declared invalid.
119   Signature      : assertion : (unit > bool) > unit = <fun>.
120   Parameter(s)   : The predicate to check.
121   Precondtion    : None.
122   Postcondtion   : Check if the predicate is valid.
123   Result         : None.
124 *)
125 let assertion f =  match f() with
126   | false -> let _ = failure in ()
127   | true ->  continue();;
128
129
130 (*
```

```
(*
  Forks the execution to 2 versions. In each of these versions, the function
  returns a different boolean. The parent execution is valid if only the child
  executions are valid.
  Signature     : forall_bool : unit > bool = <fun>.
  Parameter(s)  : None.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : Return true if the child executions are valid.
*)

let forall_bool () = Delimcc.shift prompt0 (fun x -> (x false) && (x true))
;;


(*
  Forks the execution to 2 versions. In each of these versions, the function
  returns a different boolean. The parent execution is valid if at least one of
  the child executions is valid.
  Signature     : forsome_bool : unit > bool = <fun>.
  Parameter(s)  : None.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : Returns true if at least one of the child executions is valid.
*)
let forsome_bool () = Delimcc.shift prompt0 (fun x -> (x false) || (x true))
;;

(*
  Generalizes forall_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature     : forall : a Flux. t > a = <fun>.
  Parameter(s)  : a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : return 'a.
*)
let rec forall_subFunction stream_2 =
    match (Flux.uncons stream_2) with
      | None ->
        failwith "Empty Stream"
      | Some(h, t) when Flux.isEmpty t ->
          (h, forall_bool())
      | Some(h, t) when  forall_bool() && snd(forall_subFunction t) ->
          (h, true)
      | Some(h, _) ->
          (h, false);;

let rec forall stream =
  let (h, bool) = forall_subFunction stream in
    if bool = true then
        h
    else
      match (Flux.uncons stream) with
        | None -> failwith "Empty Stream"
        | Some(h, t) when Flux.isEmpty t -> h
        | Some(h, t) -> forall t
;;

(*
  Generalizes forsome_bool and forks the current execution in as many versions
  as there are elements in the stream.
  Signature     : forsome : a Flux. t > a = <fun>.
  Parameter(s)  : a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
```

```
  Result        : return 'a.
*)
let rec forsome_subFunction stream2 =
    match (Flux.uncons stream2) with
      | None ->
        failwith "Empty Stream"
      | Some(h, t) when Flux.isEmpty t ->
          (h, forall_bool())
      | Some(h, t) when  snd(forall_subFunction t) && forall_bool() ->
          (h, true)
      | Some(h, _) ->
          (h, false);;

let rec forsome stream =
  let (h, bool) = forsome_subFunction stream in
    if bool = true then
      h
    else
      match (Flux.uncons stream) with
        | None -> failwith "Empty Stream"
        | Some(h, t) when Flux.isEmpty t -> h
        | Some(h, t) -> forsome t
;;
(*
  Forks the current execution in as many versions as there are elements in the
  stream. The parent execution is valid if at least n child executions are
  valid. We have forsome = foratleast 1.
  Signature     : foratleast : int > a Flux. t > a = <fun>.
  Parameter(s)  : int > a Flux. t.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : return 'a.
*)

let rec foratleast n stream =
  match (n, Flux.uncons stream) with
    | (_, None) | (0, _) ->
      failwith "Incorrect foratleast arguments"
    | (1, _) ->
      forsome stream
    | (_, Some(h, t)) ->
      if forsome_bool() then
        foratleast (n - 1) t
      else
        foratleast n t

(*
  Executes a program instrumented with the primitives above.
  The result is a boolean representing the validity of the execution and can be used with
    let %test
  Signature     : check : ( unit > unit) > bool.
  Parameter(s)  : function f : unit -> unit.
  Precondtion   : None.
  Postcondtion  : None.
  Result        : Return true if the execution is valid, otherwise false.
*)
let check func = Delimcc.push_prompt prompt0 (fun () -> forall_bool( func() ));;

(* ----------------------------------------EXTENSION
    ---------------------------------------- *)

(*forall_length : int Flux. t > (unit > a) > a list *)
let rec forall_length lengths f =
  match Flux.uncons lengths with
  | None ->failwith "Empty forall_length stream"
```

```
  | Some(h, t) when h = 0 -> forall_length t f
  | Some(h, t) when Flux.isEmpty t ->
    if forall_bool() then
      [f()]
    else
      []
  | Some(h, t) ->
    if forall_bool() then
      f()::forall_length t f
    else
      []
;;


(*forsome_length : int Flux. t > (unit > a) > a list *)
let rec forsome_length lengths f =
  match Flux.uncons lengths with
  | None -> failwith "Empty forsome_length stream"
  | Some(h, t) when h = 0 -> forsome_length t f
  | Some(h, t) when Flux.isEmpty t -> if forsome_bool() then [f()] else []
  | Some(h, t) ->
    if forsome_bool() then
      f()::forsome_length t f
    else
      forsome_length t f
;;


(*foratleast_length : int > int Flux. t > (unit > a) > a list *)
let rec foratleast_length n stream =
  match (n, Flux.uncons stream) with
    | (_, None) | (0, _) ->
      failwith "Incorrect foratleast_length arguments"
    | (1, _) ->
      forsome_length stream
    | (_, Some(h, t)) ->
      if forsome_bool() then
        foratleast_length (n - 1) t
      else
        foratleast_length n t
;;
```

5.2 Tests

```
open Main

let rec pgcd a b=
  if a >= b then
    if b = 0 then
      a
    else
      pgcd b (a mod b)
  else
    pgcd b a
;;

let premiers_entre_eux a b  =
  if pgcd a b = 1 then
    true
  else
    false;;

let premier n =
  let rec test_local(t,k) =
    if t*t > k then
      true
    else
      if k mod t = 0 then
        false
      else
        test_local(t+1,k)
  in test_local(2,n)
;;

let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 20 then None else Some (cpt, cpt+1)) 1 in
  check (fun () ->
  let a = forall values in
  let b = forall values in
  assumption (fun () -> premiers_entre_eux a b );
  let r = pgcd a b in
  assertion (fun () -> r = 1))
;;

let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt+1)) 2 in
  check (fun () ->
  let a = forall values in
  let b = forsome values in
  assumption (fun () -> a <> b);
  let r = premier a in
  assertion (fun () -> r || (a mod b = 0)))
;;

let pred x y z = x = 5 * y + 7 * z ;;
let %test _ =
  let interval a b = Flux. unfold (fun cpt ->  if cpt > b then None else Some (cpt, cpt+1))
    a in
  check (fun () ->
    begin
      let x = forall ( interval 10 50) in
      assumption (fun () ->  x mod 2 = 0);
      let y = forsome ( interval 0 20) in
      let z = forsome ( interval 0 20) in
      assertion (fun () ->  pred x y z)
    end
```

```
  )
;;

(* a. [|1 , 10|],  b. [|11, 20|] a <> b *)
let %test _ =
  let values1 = Flux.unfold (fun cpt -> if cpt > 10 then None else Some (cpt, cpt + 1)) 1 in
  let values2 = Flux.unfold (fun cpt -> if cpt > 20 then None else Some (cpt, cpt + 1)) 11
    in
  check(fun() ->
    let a = forall values1 in
    let b = forall values2 in
    assumption (fun () -> a <> b);
    assertion (fun() -> a <> b)
  )
  ;;

(* a,b. [|1 , 50|], a < b => a+1 < b+1 *)
let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
  check(fun() ->
    let a = forall values in
    let b = forall values in
    assumption (fun () -> a < b);
    assertion (fun() -> (a+1) < (b+1))
  )
  ;;

(* a,. [|1 , 50|], a mod 2 = 0 => a+1 mod 2 = 1 *)
let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
  check(fun() ->
    let a = forall values in
    assumption (fun () -> (a mod 2) = 0);
    assertion (fun() -> (a + 1 mod 2 = 1))
  )
  ;;

let primeFactors n =
    let rec aux d n =
      if n = 1 then [] else
        if n mod d = 0 then d :: aux d (n / d) else aux (d+1) n
    in aux 2 n;;

(* a,. [|2 , 100|], a not premier => primeFactors(a).length >= 1 *)
let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 100 then None else Some (cpt, cpt + 1)) 2 in
  check(fun() ->
    let a = forall values in
    assumption (fun () -> not(premier a));
    let facteursPremiers = primeFactors a in
    assertion (fun() -> List.length facteursPremiers > 1)
  )
  ;;

(* a,. [|1 , 50|], pgcd a b = 1 => premier_entre_eux a b *)
let %test _ =
  let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
  check(fun() ->
    let a = forall values in
    let b = forall values in
    assumption (fun () -> pgcd a b = 1);
    assertion (fun() -> premiers_entre_eux a b)
  )
  ;;

```

```
(* Merge Sort *)
let rec merge = function
  | list, []
  | [], list -> list
  | h1::t1, h2::t2 ->
      if h1 <= h2 then
        h1 :: merge (t1, h2::t2)
      else
        h2 :: merge (h1::t1, t2)
and halve = function
  | []
  | [_] as t1 -> t1, []
  | h::t ->
      let t1, t2 = halve t in
        h::t2, t1
and mergesort = function
  | []
  | [_] as list -> list
  | list ->
      let l1, l2 = halve list in
        merge (mergesort l1, mergesort l2)
;;

let %test _=
  let lengths = Flux. unfold (fun cpt ->  if cpt > 5 then None else Some (cpt, cpt+1)) 0 in
  let values = Flux. unfold (fun cpt ->  if cpt > 20 then None else Some (cpt, cpt+1)) 1 in
  check (fun () ->
  let l = forall_length lengths (fun () ->  forall values ) in
  assertion (fun () ->  List.sort Pervasives.compare l = mergesort l ))
;;

let %test _ =
  let lengths = Flux.unfold (fun cpt -> if cpt > 5 then None else Some (cpt, cpt + 1)) 0 in
  let values = Flux.unfold (fun cpt -> if cpt > 10 then None else Some (cpt, cpt + 1)) 1 in
  check (fun () ->
  let l = forall_length lengths (fun () ->  forall values ) in
  let x = forall values in
  let l1 = forsome_length lengths (fun () ->  forsome values) in
  let l2 = forsome_length lengths (fun () ->  forsome values) in
  let r = List . mem x l in
  assertion (fun () ->  r = ( l = l1@(x::l2 )))
  );;
```