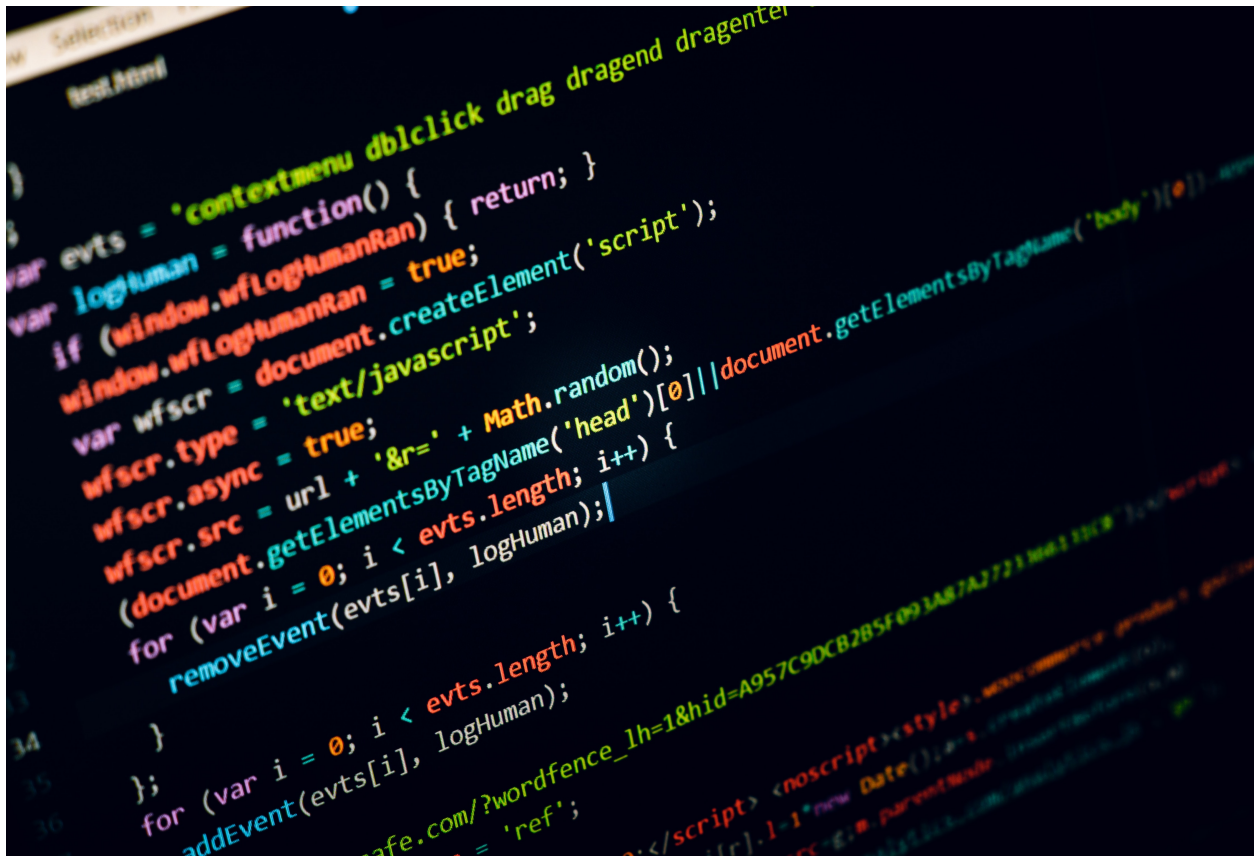


## Infrastructure de Tests - Rapport



HAMZA MOUDDENE  
YOUNES SAOUDI  
OUSSAMA ECHCHERQAOU  
MANAL HAJJI

January 24, 2021



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Architecture</b>	<b>1</b>
<b>3</b>	<b>Conception</b>	<b>1</b>
3.1	Fonctionnalités minimum . . . . .	1
3.2	Extension . . . . .	5
3.3	Test . . . . .	5
<b>4</b>	<b>Conclusion</b>	<b>5</b>
<b>5</b>	<b>Annexe</b>	<b>6</b>
5.1	Infrastructure de tests . . . . .	6
5.2	Tests . . . . .	14

# 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* :  $(\text{unit} \rightarrow \text{bool}) \rightarrow \text{unit}$ , 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.
  Precondition   : Check if the predicate is valid.
  Postcondition  : 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.
  Precondition   : None.
  Postcondition  : 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.
  Precondition   : None.
  Postcondition  : 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.

```

(*
    Interrupts the execution and makes it invalid.
    Signature    : failure : unit -> 'a = <fun>.
    Parameter(s) : None.
    Precondition : None.
    Postcondition : None.
    Result       : None.
*)
let failure () = Delimcc.shift prompt0 (fun x -> x Invalid);;

```

Figure 5: La fonction assertion

- forall bool : unit  $\rightarrow$  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.

```

(*
    Interrupts the execution and makes it invalid.
    Signature    : failure : unit -> 'a = <fun>.
    Parameter(s) : None.
    Precondition : None.
    Postcondition : None.
    Result       : None.
*)
let failure () = Delimcc.shift prompt0 (fun x -> x Invalid);;

```

Figure 6: La fonction forall\_bool

- forsome bool : unit  $\rightarrow$  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.

```

(*
    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.
    Precondition : None.
    Postcondition : None.
    Result       : Returns true if at least one of the child executions is valid.
*)
val forsome_bool : unit -> bool

```

Figure 7: La fonction forsome\_bool

- forall : 'a Flux. t  $\rightarrow$  '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.
  Precondition : None.
  Postcondition : 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.
  Precondition : None.
  Postcondition : 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 a `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.
  Precondition : None.
  Postcondition : 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.
  Precondition   : None.
  Postcondition  : 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

```
1 (* interfaces des flux utiles pour toute la séance *)
2
3 module type Iter =
4 sig
5   type 'a t
6   val vide : 'a t
7   val cons : 'a -> 'a t -> 'a t
8   val uncons : 'a t -> ('a * 'a t) option
9   val unfold : ('s -> ('a * 's) option) -> 's -> 'a t
10  val filter : ('a -> bool) -> 'a t -> 'a t
11  val append : 'a t -> 'a t -> 'a t
12  val constant : 'a -> 'a t
13  val map : ('a -> 'b) -> 'a t -> 'b t
14  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
15  val apply : ('a -> 'b) t -> 'a t -> 'b t
16  val isEmpty : 'a t -> bool
17 end
18
19
20 (* Module Flux implantant l'interface de flux Iter *)
21 (* a l'aide d'une structure de donnees paresseuse *)
22 type 'a flux = Tick of ('a * 'a flux) option Lazy.t;;
23 module Flux : Iter with type 'a t = 'a flux;;
24 type result;;
25
26 (*
27   Filters and continues just the executions that check the given predicate in
28   parameter. The other executions should be stopped and declared valid.
29   Signature      : assumption : (unit -> bool) -> unit = <fun>.
30   Parameter(s)   : The predicate to check.
31   Precondition   : Check if the predicate is valid.
32   Postcondition   : None.
33   Result         : None.
34 *)
35 val assumption : (unit -> bool) -> unit
36
37 (*
38   Filters and continues just the executions that check the given predicate in
39   parameter. The other executions should be stopped and declared invalid.
40   Signature      : assertion : (unit -> bool) -> unit = <fun>.
41   Parameter(s)   : The predicate to check.
42   Precondition   : None.
43   Postcondition   : Check if the predicate is valid.
44   Result         : None.
45 *)
46 val assertion : (unit -> bool) -> unit
47
48 (*
49   Interrupt the execution and makes it valid.
50   Signature      : miracle : unit -> 'a = <fun>.
51   Parameter(s)   : None.
52   Precondition   : None.
53   Postcondition   : None.
54   Result         : None.
55 *)
56 val miracle : unit -> result
57
58 (*
59   Interrupts the execution and makes it invalid.
```

```

60     Signature      : failure : unit > 'a = <fun>.
61     Parameter(s)   : None.
62     Precondtion    : None.
63     Postcondtion   : None.
64     Result         : None.
65 *)
66 val failure : unit -> result
67
68 (*
69     Forks the execution to 2 versions. In each of these versions, the function
70     returns a different boolean. The parent execution is valid if only the child
71     executions are valid.
72     Signature      : forall_bool : unit > bool = <fun>.
73     Parameter(s)   : None.
74     Precondtion    : None.
75     Postcondtion   : None.
76     Result         : Return true if the child executions are valid.
77 *)
78 val forall_bool : unit -> bool
79
80 (*
81     Forks the execution to 2 versions. In each of these versions, the function
82     returns a different boolean. The parent execution is valid if at least one of
83     the child executions is valid.
84     Signature      : forsome_bool : unit > bool = <fun>.
85     Parameter(s)   : None.
86     Precondtion    : None.
87     Postcondtion   : None.
88     Result         : Returns true if at least one of the child executions is valid.
89 *)
90 val forsome_bool : unit -> bool
91
92 (*
93     Generalizes forall_bool and forks the current execution in as many versions
94     as there are elements in the stream.
95     Signature      : forall : 'a Flux. t > 'a = <fun>.
96     Parameter(s)   : 'a Flux. t.
97     Precondtion    : None.
98     Postcondtion   : None.
99     Result         : return 'a.
100 *)
101 val forall : 'a Flux.t -> 'a
102
103 (*
104     Generalizes forsome_bool and forks the current execution in as many versions
105     as there are elements in the stream.
106     Signature      : forsome : 'a Flux. t > 'a = <fun>.
107     Parameter(s)   : 'a Flux. t.
108     Precondtion    : None.
109     Postcondtion   : None.
110     Result         : return 'a.
111 *)
112 val forsome : 'a Flux.t -> 'a
113
114 (*
115     Forks the current execution in as many versions as there are elements in the
116     stream. The parent execution is valid if at least n child executions are
117     valid. We have forsome = foratleast 1.
118     Signature      : foratleast : int > 'a Flux. t > 'a = <fun>.
119     Parameter(s)   : int > 'a Flux. t.
120     Precondtion    : None.
121     Postcondtion   : None.
122     Result         : return 'a.
123 *)
124 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   Precondition   : None.
131   Postcondition  : 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

```

```

1 module type Iter =
2 sig
3   type 'a t
4   val vide : 'a t
5   val cons : 'a -> 'a t -> 'a t
6   val uncons : 'a t -> ('a * 'a t) option
7   val unfold : ('s -> ('a * 's) option) -> 's -> 'a t
8   val filter : ('a -> bool) -> 'a t -> 'a t
9   val append : 'a t -> 'a t -> 'a t
10  val constant : 'a -> 'a t
11  val map : ('a -> 'b) -> 'a t -> 'b t
12  val map2 : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
13  val apply : ('a -> 'b) t -> 'a t -> 'b t
14  val isEmpty : 'a t -> bool
15 end
16
17 type 'a flux = Tick of ('a * 'a flux) option Lazy.t;;
18
19 module Flux : Iter with type 'a t = 'a flux =
20 struct
21   type 'a t = 'a flux = Tick of ('a * 'a t) option Lazy.t;;
22
23   let vide = Tick (lazy None);;
24
25   let cons t q = Tick (lazy (Some (t, q)));;
26
27   let uncons (Tick flux) = Lazy.force flux;;
28
29   let rec apply f x =
30     Tick (lazy (
31       match uncons f, uncons x with
32       | None, _ -> None
33       | _, None -> None
34       | Some (tf, qf), Some (tx, qx) -> Some (tf tx, apply qf qx)));;
35
36   let rec unfold f e =
37     Tick (lazy (
38       match f e with
39       | None -> None
40       | Some (t, e') -> Some (t, unfold f e')));;
41
42   let rec filter p flux =
43     Tick (lazy (
44       match uncons flux with
45       | None -> None
46       | Some (t, q) -> if p t then Some (t, filter p q)
47                          else uncons (filter p q)));;
48
49   let rec append flux1 flux2 =
50     Tick (lazy (
51       match uncons flux1 with
52       | None -> uncons flux2
53       | Some (t1, q1) -> Some (t1, append q1 flux2)));;
54
55   let constant c = unfold (fun () -> Some (c, ())) ();;
56   (* implantation rapide mais inefficace de map *)
57   let map f i = apply (constant f) i;;
58
59   let map2 f i1 i2 = apply (apply (constant f) i1) i2;;
60
61   let isEmpty stream =
62     match (uncons stream) with
63     | None -> true
64     | _ -> false;;
65 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   Precondition   : None.
77   Postcondition   : 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   Precondition   : None.
87   Postcondition   : 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   Precondition   : None.
97   Postcondition   : 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   Precondition   : Check if the predicate is valid.
108   Postcondition   : 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   Precondition   : None.
122   Postcondition   : 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 (*

```

```

131 Forks the execution to 2 versions. In each of these versions, the function
132 returns a different boolean. The parent execution is valid if only the child
133 executions are valid.
134 Signature      : forall_bool : unit > bool = <fun>.
135 Parameter(s)   : None.
136 Precondition   : None.
137 Postcondition  : None.
138 Result         : Return true if the child executions are valid.
139 *)
140
141 let forall_bool () = Delimcc.shift prompt0 (fun x -> (x false) && (x true))
142 ;;
143
144
145 (*
146 Forks the execution to 2 versions. In each of these versions, the function
147 returns a different boolean. The parent execution is valid if at least one of
148 the child executions is valid.
149 Signature      : forsome_bool : unit > bool = <fun>.
150 Parameter(s)   : None.
151 Precondition   : None.
152 Postcondition  : None.
153 Result         : Returns true if at least one of the child executions is valid.
154 *)
155 let forsome_bool () = Delimcc.shift prompt0 (fun x -> (x false) || (x true))
156 ;;
157
158 (*
159 Generalizes forall_bool and forks the current execution in as many versions
160 as there are elements in the stream.
161 Signature      : forall : a Flux. t > a = <fun>.
162 Parameter(s)   : a Flux. t.
163 Precondition   : None.
164 Postcondition  : None.
165 Result         : return 'a.
166 *)
167 let rec forall_subFunction stream_2 =
168   match (Flux.uncons stream_2) with
169   | None ->
170     failwith "Empty Stream"
171   | Some(h, t) when Flux.isEmpty t ->
172     (h, forall_bool())
173   | Some(h, t) when forall_bool() && snd(forall_subFunction t) ->
174     (h, true)
175   | Some(h, _) ->
176     (h, false);;
177
178 let rec forall stream =
179   let (h, bool) = forall_subFunction stream in
180   if bool = true then
181     h
182   else
183     match (Flux.uncons stream) with
184     | None -> failwith "Empty Stream"
185     | Some(h, t) when Flux.isEmpty t -> h
186     | Some(h, t) -> forall t
187 ;;
188
189 (*
190 Generalizes forsome_bool and forks the current execution in as many versions
191 as there are elements in the stream.
192 Signature      : forsome : a Flux. t > a = <fun>.
193 Parameter(s)   : a Flux. t.
194 Precondition   : None.
195 Postcondition  : None.

```

```

196   Result          : return 'a.
197 *)
198 let rec forsome_subFunction stream2 =
199   match (Flux.uncons stream2) with
200   | None ->
201     failwith "Empty Stream"
202   | Some(h, t) when Flux.isEmpty t ->
203     (h, forall_bool())
204   | Some(h, t) when snd(forall_subFunction t) && forall_bool() ->
205     (h, true)
206   | Some(h, _) ->
207     (h, false);;
208
209 let rec forsome stream =
210   let (h, bool) = forsome_subFunction stream in
211   if bool = true then
212     h
213   else
214     match (Flux.uncons stream) with
215     | None -> failwith "Empty Stream"
216     | Some(h, t) when Flux.isEmpty t -> h
217     | Some(h, t) -> forsome t
218 ;;
219 (*
220   Forks the current execution in as many versions as there are elements in the
221   stream. The parent execution is valid if at least n child executions are
222   valid. We have forsome = foratleast 1.
223   Signature       : foratleast : int > a Flux. t > a = <fun>.
224   Parameter(s)    : int > a Flux. t.
225   Precondition    : None.
226   Postcondition   : None.
227   Result          : return 'a.
228 *)
229
230 let rec foratleast n stream =
231   match (n, Flux.uncons stream) with
232   | (_, None) | (0, _) ->
233     failwith "Incorrect foratleast arguments"
234   | (1, _) ->
235     forsome stream
236   | (_, Some(h, t)) ->
237     if forsome_bool() then
238       foratleast (n - 1) t
239     else
240       foratleast n t
241
242 (*
243   Executes a program instrumented with the primitives above.
244   The result is a boolean representing the validity of the execution and can be used with
245   let %test
246   Signature       : check : ( unit > unit ) > bool.
247   Parameter(s)    : function f : unit -> unit.
248   Precondition    : None.
249   Postcondition   : None.
250   Result          : Return true if the execution is valid, otherwise false.
251 *)
252 let check func = Delimcc.push_prompt prompt0 (fun () -> forall_bool( func() ));;
253 (* -----EXTENSION
254   ----- *)
255 (*forall_length : int Flux. t > (unit > a) > a list *)
256 let rec forall_length lengths f =
257   match Flux.uncons lengths with
258   | None -> failwith "Empty forall_length stream"

```

```

259 | Some(h, t) when h = 0 -> forall_length t f
260 | Some(h, t) when Flux.isEmpty t ->
261   if forall_bool() then
262     [f()]
263   else
264     []
265 | Some(h, t) ->
266   if forall_bool() then
267     f()::forall_length t f
268   else
269     []
270 ;;
271
272
273 (*forsome_length : int Flux. t > (unit > a) > a list *)
274 let rec forsome_length lengths f =
275   match Flux.uncons lengths with
276   | None -> failwith "Empty forsome_length stream"
277   | Some(h, t) when h = 0 -> forsome_length t f
278   | Some(h, t) when Flux.isEmpty t -> if forsome_bool() then [f()] else []
279   | Some(h, t) ->
280     if forsome_bool() then
281       f()::forsome_length t f
282     else
283       forsome_length t f
284 ;;
285
286
287 (*foratleast_length : int > int Flux. t > (unit > a) > a list *)
288 let rec foratleast_length n stream =
289   match (n, Flux.uncons stream) with
290   | (_, None) | (0, _) ->
291     failwith "Incorrect foratleast_length arguments"
292   | (1, _) ->
293     forsome_length stream
294   | (_, Some(h, t)) ->
295     if forsome_bool() then
296       foratleast_length (n - 1) t
297     else
298       foratleast_length n t
299 ;;

```



## 5.2 Tests

```
1 open Main
2
3 let rec pgcd a b =
4   if a >= b then
5     if b = 0 then
6       a
7     else
8       pgcd b (a mod b)
9   else
10    pgcd b a
11 ;;
12
13 let premiers_entre_eux a b =
14   if pgcd a b = 1 then
15     true
16   else
17     false;;
18
19 let premier n =
20   let rec test_local(t,k) =
21     if t*t > k then
22       true
23     else
24       if k mod t = 0 then
25         false
26       else
27         test_local(t+1,k)
28   in test_local(2,n)
29 ;;
30
31 let %test _ =
32   let values = Flux.unfold (fun cpt -> if cpt > 20 then None else Some (cpt, cpt+1)) 1 in
33   check (fun () ->
34     let a = forall values in
35     let b = forall values in
36     assumption (fun () -> premiers_entre_eux a b );
37     let r = pgcd a b in
38     assertion (fun () -> r = 1))
39 ;;
40
41 let %test _ =
42   let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt+1)) 2 in
43   check (fun () ->
44     let a = forall values in
45     let b = forsome values in
46     assumption (fun () -> a <> b);
47     let r = premier a in
48     assertion (fun () -> r || (a mod b = 0)))
49 ;;
50
51 let pred x y z = x = 5 * y + 7 * z ;;
52 let %test _ =
53   let interval a b = Flux.unfold (fun cpt -> if cpt > b then None else Some (cpt, cpt+1))
54     a in
55   check (fun () ->
56     begin
57       let x = forall ( interval 10 50) in
58       assumption (fun () -> x mod 2 = 0);
59       let y = forsome ( interval 0 20) in
60       let z = forsome ( interval 0 20) in
61       assertion (fun () -> pred x y z)
62     end
```

```

62 )
63 ;;
64
65 (* a. [|1 , 10|], b. [|11, 20|] a <> b *)
66 let %test _ =
67   let values1 = Flux.unfold (fun cpt -> if cpt > 10 then None else Some (cpt, cpt + 1)) 1 in
68   let values2 = Flux.unfold (fun cpt -> if cpt > 20 then None else Some (cpt, cpt + 1)) 11
69   in
70   check(fun() ->
71     let a = forall values1 in
72     let b = forall values2 in
73     assumption (fun () -> a <> b);
74     assertion (fun() -> a <> b)
75   )
76   ;;
77
78 (* a,b. [|1 , 50|], a < b => a+1 < b+1 *)
79 let %test _ =
80   let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
81   check(fun() ->
82     let a = forall values in
83     let b = forall values in
84     assumption (fun () -> a < b);
85     assertion (fun() -> (a+1) < (b+1))
86   )
87   ;;
88
89 (* a,. [|1 , 50|], a mod 2 = 0 => a+1 mod 2 = 1 *)
90 let %test _ =
91   let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
92   check(fun() ->
93     let a = forall values in
94     assumption (fun () -> (a mod 2) = 0);
95     assertion (fun() -> (a + 1 mod 2 = 1))
96   )
97   ;;
98
99 let primeFactors n =
100   let rec aux d n =
101     if n = 1 then [] else
102     if n mod d = 0 then d :: aux d (n / d) else aux (d+1) n
103   in aux 2 n;;
104
105 (* a,. [|2 , 100|], a not premier => primeFactors(a).length >= 1 *)
106 let %test _ =
107   let values = Flux.unfold (fun cpt -> if cpt > 100 then None else Some (cpt, cpt + 1)) 2 in
108   check(fun() ->
109     let a = forall values in
110     assumption (fun () -> not(premier a));
111     let facteursPremiers = primeFactors a in
112     assertion (fun() -> List.length facteursPremiers > 1)
113   )
114   ;;
115
116 (* a,. [|1 , 50|], pgcd a b = 1 => premier_entre_eux a b *)
117 let %test _ =
118   let values = Flux.unfold (fun cpt -> if cpt > 50 then None else Some (cpt, cpt + 1)) 1 in
119   check(fun() ->
120     let a = forall values in
121     let b = forall values in
122     assumption (fun () -> pgcd a b = 1);
123     assertion (fun() -> premiers_entre_eux a b)
124   )
125   ;;

```

```

126 (* Merge Sort *)
127 let rec merge = function
128 | list, []
129 | [], list -> list
130 | h1::t1, h2::t2 ->
131     if h1 <= h2 then
132         h1 :: merge (t1, h2::t2)
133     else
134         h2 :: merge (h1::t1, t2)
135 and halve = function
136 | []
137 | [_] as t1 -> t1, []
138 | h::t ->
139     let t1, t2 = halve t in
140     h::t2, t1
141 and mergesort = function
142 | []
143 | [_] as list -> list
144 | list ->
145     let l1, l2 = halve list in
146     merge (mergesort l1, mergesort l2)
147 ;;
148
149 let %test _ =
150     let lengths = Flux.unfold (fun cpt -> if cpt > 5 then None else Some (cpt, cpt+1)) 0 in
151     let values = Flux.unfold (fun cpt -> if cpt > 20 then None else Some (cpt, cpt+1)) 1 in
152     check (fun () ->
153         let l = forall_length lengths (fun () -> forall values ) in
154         assertion (fun () -> List.sort Pervasives.compare l = mergesort l ))
155 ;;
156
157 let %test _ =
158     let lengths = Flux.unfold (fun cpt -> if cpt > 5 then None else Some (cpt, cpt + 1)) 0 in
159     let values = Flux.unfold (fun cpt -> if cpt > 10 then None else Some (cpt, cpt + 1)) 1 in
160     check (fun () ->
161         let l = forall_length lengths (fun () -> forall values ) in
162         let x = forall values in
163         let l1 = forsome_length lengths (fun () -> forsome values) in
164         let l2 = forsome_length lengths (fun () -> forsome values) in
165         let r = List . mem x l in
166         assertion (fun () -> r = ( l = l1@(x::l2 )))
167     );;

```