



**Rapport infrastructure de test**  
Projet Programmation Fonctionnelle 2SN -  
Systèmes Logiciels - Session 1 - 2020/2021

Imad Abakarim  
Omar Sabri  
Mouad Dahhoumi  
Souhail Amghar

Département Sciences Numériques - Deuxième année  
Novembre 2020

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Fonctionnalités minimum</b>	<b>3</b>
<b>3</b>	<b>Choix et conception</b>	<b>4</b>
3.1	Primitive forall_bool . . . . .	4
3.2	Primitive forall_bool_x_y . . . . .	5
3.3	Primitive forall . . . . .	5
3.3.1	Idée ratée . . . . .	5
3.3.2	Idée réussie . . . . .	5
3.4	Primitive foratleast . . . . .	5
3.5	Primitive forsome . . . . .	5
<b>4</b>	<b>Tests</b>	<b>6</b>
4.1	Etape pour tester . . . . .	6
<b>5</b>	<b>Extensions</b>	<b>6</b>
5.1	Tests . . . . .	6
<b>6</b>	<b>Raffinement du programme</b>	<b>7</b>

## 1 Introduction

On 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 (universellement ou existentiellement) 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 Fonctionnalités minimum

Dans toutes les parties du projet, on utilisera la monade flux pour représenter la structure des données itératives.

```

module type MONADE =
sig
  type 'a t
  val map : ('a -> 'b) -> ('a t -> 'b t)
  val return : 'a -> 'a t
  val (»=) : 'a t -> ('a -> 'b t) -> 'b t
  val zero : 'a t
  val (++) : 'a t -> 'a t -> 'a t
  val uncons : 'a t -> ('a 'a t) option
  val unfold : ('b -> ('a 'b) option) -> ('b -> 'a t)
end

module NDIter : MONADE =
struct
  type 'a t = Tick of ('a*'a t) option Lazy.t
  let uncons (Tick f) = Lazy.force f
  let rec map f flux = Tick(lazy(match uncons flux with
    |None->None
    |Some(t,q) -> Some(f t,map f q )))
  let return a = Tick(lazy(Some(a,Tick(lazy(None)))))
  let rec (++) f1 f2 = Tick(lazy(match uncons f1 with
    |None->uncons f2
    |Some(t,q)->Some(t,q ++ f2)))
  let rec (»=) flux f = Tick(lazy(match uncons flux with
    |None->None

```

```
|Some(t,q)->uncons(f t ++ (q»=f)))
let zero = Tick(lazy(None))
let rec unfold f e =
  Tick(lazy(
    match f e with
    |None->None
    |Some(t,e')-> Some(t,unfold f e')));;
end
```

Dans cette partie, on implante les primitives figurées dans l'interface ci-dessous :

```
module InfraTest :
sig
type 'a t = 'a NDIter.t
val suivre : bool -> bool -> int -> bool
val suivreAll : bool t -> bool
val failure : unit -> unit
val suivreAtleast : int -> bool t -> bool
val forall_bool : unit -> bool
val forsomes_bool : unit -> bool
val forall_bool_x_y : 'a * 'a -> 'a
val miracle : unit -> unit
val forall : 'a t -> 'a
val foratleast : int -> 'a t -> 'a
val forsomes : 'a t -> 'a
val assumption : (unit -> bool) -> unit
val assertion : (unit -> bool) -> unit
val check : (unit -> 'a) -> bool
end
```

La sémantique et l'implantation de ces méthodes sont dans le fichier *projet.ml*.

## 3 Choix et conception

### 3.1 Primitive forall\_bool

```
val forall_bool : unit -> bool
```

Le principe est le suivant : On commence par capturer ce qui reste à exécuter dans une continuation en utilisant un `shift`, puis relancer deux exécutions en renvoyant `{true : pour la première, false : pour la deuxième}`. Ces deux exécutions sont encapsulées dans une fonction *suivre* qui renvoie `true`, si les deux exécutions sont valides, et `false` sinon.

De même pour `forsome_bool`.

### 3.2 Primitive `forall_bool_x_y`

```
val forall_bool_x_y : 'a * 'a -> 'a
```

Même que `forall_bool` mais on renvoie deux valeurs passées en paramètre pour l'exécution.

### 3.3 Primitive `forall`

```
val forall : 'a t -> 'a
```

#### 3.3.1 Idée ratée

On avait l'idée d'utiliser `forall_bool_x_y` pour implanter la primitive `forall` de la manière suivante :

On prend deux valeurs du flux `x'` et `y'`, on lance l'exécution `forall_bool_x_y x' y'`, si l'exécution est valide, on continue en prenant les deux valeurs qui suivent dans le flux, sinon on exécute la primitive `failure`. Malheureusement, on a pas pu implanter cette idée, car, en effectuant des tests, on remarque que lors de l'appel de la primitive `forall_bool_x_y`, la continuation est capturée localement, et non pas du niveau de la procédure du test.

#### 3.3.2 Idée réussie

Une autre idée consiste à capturer la continuation, puis lancer l'exécution autant de fois qu'il y a des valeurs dans le flux, puis capturer le résultat de toutes exécutions en utilisant `suivreAll` qui renvoie `true` si toutes les exécutions sont réussies.

### 3.4 Primitive `foratleast`

```
val foratleast : int -> 'a t -> 'a
```

L'implantation consiste à capturer la continuation, puis lancer l'exécution autant de fois qu'il y a des valeurs dans le flux, puis capturer le résultat de toutes les exécutions en utilisant `suivreAtLeaste` qui renvoie `true` si au moins `n` exécutions sont valides.

### 3.5 Primitive `forsome`

```
val forsome : 'a t -> 'a
```

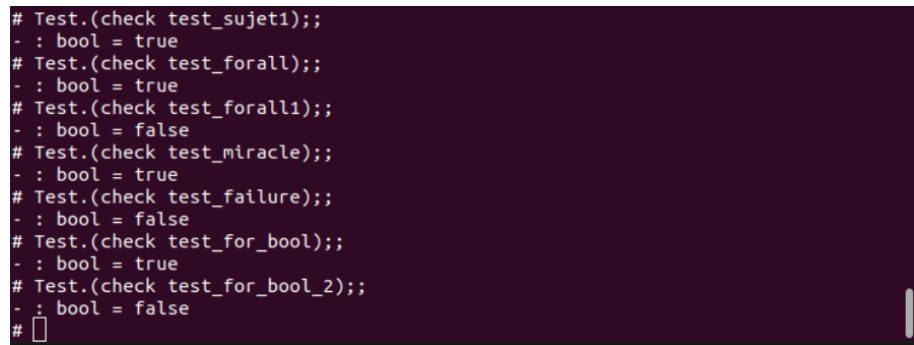
```
forsome = foratleast 1
```

## 4 Tests

Pour tester ces primitives, on définit des programmes qui seront exécutés au sein d'un check. Voir le fichier *projet.ml*.

### 4.1 Etape pour tester

On lance la commande `ocaml -I ocamlfind query delimcc delimcc.cma`. On copie le fichier *projet.ml* dans le terminal, puis, dans le shell `ocaml`, on exécute la fonction `ModuleTest.(check prog_int);;`



```
# Test.(check test_sujet1);;
- : bool = true
# Test.(check test_forall);;
- : bool = true
# Test.(check test_forall1);;
- : bool = false
# Test.(check test_miracle);;
- : bool = true
# Test.(check test_failure);;
- : bool = false
# Test.(check test_for_bool);;
- : bool = true
# Test.(check test_for_bool_2);;
- : bool = false
#
```

FIGURE 1 – Capture test

## 5 Extensions

On choisit d'implanter la deuxième extension.

```
module QuantifList =
sig
val construire : int -> (unit -> 'a) -> 'a list
val forsome_length : int t -> (unit -> 'a) -> 'a list
val forall_length : int t -> (unit -> 'a) -> 'a list
end
```

L'implantation de cette interface est dans le fichier *projet.ml*.

### 5.1 Tests

On teste notre module en utilisant les tests fournis dans le projet.

```
# TestQuantif.(check test_sujet);;  
- : bool = true  
#
```

FIGURE 2 – Capture test

**Remarque :** Ce test est fait en quantifiant sur les listes de taille 3 et en prenant des valeurs dans l'intervalle  $[0, 4]$ . Donc, on a  $5 + 5^2 + 5^3$  listes possibles (exécutions). On remarque que la complexité est exponentielle.

## 6 Raffinement du programme

On teste en utilisant le test de la partie 3 du projet. L'exécution prend beaucoup de temps, ce qui confirme la remarque précédente.