Rendu du Projet
Programmation Fonctionnelle
Conception d'une API de *property testing*

Date du rendu : Dimanche 2 Avril

Réalisé par :
Pierre de Vaugiraud
Giovanni Fieux
Louis Grandel
Léo Laiolo
Nicolas Lamarque

ING2 GSI groupe 1

Matière : Programmation Fonctionnelle
Professeur : Mme Inès Alaya

Pôle Informatique
Année Scolaire : 2022/23

Sommaire

Introduction

Ce document est un rapport exhaustif de notre projet d'API de property testing. L'objectif était de concevoir un générateur de tests en Ocaml. Pour cela, des modules ainsi que leurs fonctionnalités nous étaient fournis. Nous devions coder ces modules. Un fichier d'exemples de tests nous a été transmis afin de vérifier le bon fonctionnement de notre implémentation.

Dans les sections qui suivent, nous décrirons le rôle de chaque module, les descriptions des difficultés rencontrées, une analyse de la solution fournie ainsi que des exemples d'application.

Module Property

```
module Property :
  sig
    (** Type d'une propriété portant sur des éléments de type 'a
      * Une propriété est une fonction booléenne. *)
    type 'a t = 'a -> bool

    (* CONSTANTES *)

    (** Propriété toujours vérifiée *)
    val always_true  : 'a t

    (** Propriété jamais   vérifiée *)
    val always_false : 'a t
```

Description succincte de la solution mise en place

Le module Property est conçu pour représenter et manipuler des propriétés, qui sont des fonctions booléennes d'un seul argument. Ce module implémente deux constantes, always_true et always_false, qui représentent des propriétés qui renvoient toujours vrai et toujours faux, respectivement. Le module Property fournit également une fonction combine (fonctionnalité supplémentaire) pour combiner deux propriétés en une nouvelle propriété qui est vraie si les deux propriétés d'origine sont vraies.

```
    (* Fonction qui combine deux propriétés en une nouvelle propriété qui est vraie si les deux
propriétés sont vraies. *)
    let combine (p1 : 'a t) (p2 : 'a t) : 'a t = fun x -> p1 x && p2 x
```

Difficultés rencontrées et leur résolution

Le code donné ne présente pas de difficultés particulières. Cependant, il est important de souligner que le module Property est plutôt simple et pourrait être étendu avec des fonctionnalités supplémentaires pour rendre la manipulation de propriétés plus puissante et flexible.

Analyse de la pertinence des fonctionnalités, des avantages et des limites

Le module Property offre des fonctionnalités de base pour travailler avec des propriétés, ce qui est un élément essentiel du Property Testing. L'avantage de cette approche est qu'elle fournit un moyen simple et intuitif de décrire les propriétés qui doivent être vérifiées.

Cependant, il y a aussi des limites. Par exemple, le module Property ne permet pas de représenter des propriétés dépendant de plusieurs arguments. De plus, la gestion des propriétés dépendant du temps (par exemple, des propriétés qui changent avec le temps) n'est pas supportée.

Exemple d'application du module Property

Voici un exemple concret pour vérifier la **fonctionnalité supplémentaire** *combine* du module Property. Ce code teste si la propriété combinée est satisfaite pour 1000 paires d'entiers générés aléatoirement. La propriété combinée est satisfaite si la somme de deux entiers est positive et paire. Le test peut réussir ou échouer en fonction des valeurs générées. Si le test échoue, cela signifie qu'il existe au moins une paire d'entiers pour laquelle la somme n'est pas à la fois positive et paire.

```
open Property
open Generator
open Reduction
open Test

(* Définir les propriétés *)
let positive_sum_prop : (int * int) Property.t = fun (x, y) -> x + y > 0
let even_sum_prop : (int * int) Property.t = fun (x, y) -> (x + y) mod 2 = 0

(* Combiner les propriétés *)
let combined_prop = Property.combine positive_sum_prop even_sum_prop

(* Créer un générateur de paires d'entiers *)
let int_gen = Generator.int (-100) 100
let pair_gen = Generator.combine int_gen int_gen

(* Définir une stratégie de réduction
let int_red = Reduction.int
let pair_red = Reduction.combine int_red int_red *)

(* Créer un test *)
let test = Test.make_test pair_gen pair_red combined_prop

(* Effectuer le test *)
let n = 1  (* Nombre de valeurs à tester *)
let result = Test.check n test

(* Afficher le résultat *)
let () = if result
         then Printf.printf "Le test a réussi.\n"
         else Printf.printf "Le test a échoué.\n"
```

Module Generator

```
module Generator :
  sig
    (** Type du générateur pseudo-aléatoire de données de type 'a *)
    type 'a t

    (** Renvoie une nouvelle valeur aléeatoire
      * @param gen générateur pseudo-aléatoire
      * @return    nouvelle valeur aléatoire en utilisant `gen`
      *)
    val next : 'a t -> 'a

    (** Générateur constant
```

Description succincte de la solution mise en place

Le module Generator permet de générer pseudo-aléatoirement des valeurs, des caractères simples ou alphanumériques, des chaînes de caractères ou des listes. Pour cela, nous avons créé un objet Generator ayant la forme d'une fonction prenant en argument une graine et retournant une valeur aléatoire.

Difficultés rencontrées et leur résolution

Nous avons rencontré quelques difficultés en réalisant ce module. En effet, nous avions du mal à comprendre comment devaient fonctionner les générateurs. D'autant plus que leur structure n'était pas définie.
Nous nous sommes donc inspirés des générateurs pseudo-aléatoires fournis par la bibliothèque Random. Nous avons compris qu'un générateur devait être fonction afin de pouvoir créer de nouvelles valeurs à chaque utilisation. Nous avons donc décidé de lui donner comme paramètre une "graine" (ou "seed" en anglais) à laquelle une valeur précise sera attribuée. En utilisant deux fois la même graine, on obtient deux fois la même valeur, d'où le nom de "générateur pseudo-aléatoire".

Analyse de la pertinence des fonctionnalités, des avantages et des limites

Les générateurs réalisés permettent d'obtenir une grande diversité de données. Ils sont notamment plus nombreux et plus riches que ceux proposés par la bibliothèque Random d'Ocaml.

Cependant, la fonctionnalité filter n'est pas très efficace. Cette méthode permet de générer des valeurs respectant une fonction de filtre mais ne renvoie pas d'option en cas d'impossibilité de trouver une donnée compatible. Nous avons codé cette méthode de façon à ce qu'elle génère 1000 valeurs. Si elle trouve une donnée respectant la condition passée en argument, elle la retourne, sinon, elle renvoie une donnée aléatoire indépendamment du filtre.
Il aurait été plus judicieux que cette méthode retourne des types option afin d'utiliser le None en cas d'incompatibilité entre le filtre et le générateur.

Exemple d'application du module Generator

Voici quelques exemples d'utilisation du module Generator, contenus dans GeneratorApplication, ci-dessous :

```
(** Exemples d'utilisation**)
let gen_int = Generator.int 1 30;;
Generator.next gen_int;;

let gen_intnonneg = Generator.int_nonneg 50;;
Generator.next gen_intnonneg;;

let gen_string = Generator.string 8 Generator.char;;
Generator.next gen_string;;

let gen_char = Generator.char;;
Generator.next gen_char;;

let gen_list = Generator.list 8 Generator.char;;
Generator.next gen_list;;
```

On crée des générateurs pseudo-aléatoires d'entiers entre 1 et 30, d'entiers non négatifs inférieurs à 50, de chaînes contenant 8 caractères, de caractères et de listes de caractères avec une taille de 8 éléments. On appelle la méthode next du générateur pour générer pseudo-aléatoirement les valeurs.

Module Réduction

```
module Reduction :
  sig
    (** Type d'une stratégie de réduction des éléments de type 'a
      * Une stratégie associe à chaque valeur une liste de propositions plus "simples".
      * NB : Les propositions sont ordonnées dans l'ordre croissance de "simplicité"
      *      (i.e. les valeurs les plus "simples" sont en début de liste).
      * IMPORTANT : Les stratégies implémentées respectent les conditions des
générateurs correspondants.
      *)
    type 'a t = 'a -> 'a list

    (** La stratégie vide : ne renvoie aucune proposition de réduction *)
    val empty : 'a t

    (* TYPES DE BASE *)

    (** Stratégie de réduction sur les entiers
      * @param n entier
      * @return  liste d'entiers plus "simples" entre `-|n|` et `|n|`
      *)
    val int : int t

    (** Stratégie de réduction sur les entiers positifs
      * @param n entier positif
      * @return  liste d'entiers naturels plus "simples" entre 0 et `n`
      *)
    val int_nonneg : int t
```

Description succincte de la solution mise en place

Le Module Réduction a pour but de récupérer les tests générés par le Module Generator et de générer d'autres données "plus simples" pour trouver les tests les plus élémentaires possible. Les différentes stratégies de réduction sont appliquées sur différents types d'entrées: les caractères alphanumériques, strings, integers et floats.
Ce module gère aussi les couples de variables et peut utiliser un filtre sous la forme d'une propriété et renvoyer les stratégies de réduction vérifiant la propriété en question.

Difficultés rencontrées et leur résolution

La première difficulté rencontrée a été la compréhension de la consigne sur les stratégie de réduction qui sont censées "simplifier" les données en entrée. En effet, sans contexte d'utilisation de l'API il était compliqué de se décider sur quels genres de stratégie appliquer. Par exemple, il est difficile de simplifier un caractère seul sans savoir ce qui le rend "simple", après discussion avec notre professeure à ce sujet et nous avons décidé que dans ce cas aucune réduction n'est appliquée et on renvoie le caractère.

Une autre difficulté a été la consigne de classer les résultats obtenus après réduction dans les fonction string ou list par ordre de "simplicité", n'ayant aucune précision pour quantifier la simplicité dans le sujet et sans contexte d'utilisation il nous est nous n'avons aucun moyen d'établir ce critère de "simplicité" donc nous renvoyons les résultats tels quels.

Analyse de la pertinence des fonctionnalités, des avantages et des limites

L'approche de ce module Reduction est intéressante sur plusieurs aspects:

- Réduction efficace des contre-exemples : La réduction consiste à prendre un contre-exemple qui viole une propriété donnée et à le réduire à sa taille minimale tout en conservant sa violation de propriété. Le module Reduction permet de réaliser cette opération de manière efficace, ce qui facilite le processus de correction des erreurs.
- Génération de cas de test plus efficace : Le module Reduction permet également d'améliorer l'efficacité de la génération de cas de test. Il permet de détecter plus rapidement les erreurs dans le code en générant des cas de test plus petits et plus simples, tout en conservant leur validité.
- Meilleure compréhension des erreurs : La réduction des contre-exemples permet de mieux comprendre les erreurs dans le code et de les diagnostiquer plus facilement. En effet, les contre-exemples réduits sont plus simples à analyser et à comprendre que les contre-exemples originaux, ce qui facilite la tâche des développeurs dans le processus de débogage.

Cependant la mise en place de ce genre de solution est souvent complexe à mettre en place et il faut une bonne maîtrise technique pour l'implémenter correctement.
De plus, la réduction des contre-exemples peut ne pas toujours être possible pour certains types d'erreurs ou de propriétés. Dans ces cas, le module Reduction peut ne pas être en mesure de fournir des résultats significatifs pour la génération de cas de test.

Enfin la réduction des contre-exemples peut être une tâche intensive en termes de ressources, nécessitant une puissance de calcul et une mémoire supplémentaires pour réaliser efficacement l'opération de réduction.

Exemple d'application du module Réduction

Voici quelques exemples d'utilisation du module Reduction.

```
(* Création d'un objet "red_2_float" de type Reduction avec la valeur flottante 2.0 *)
let red_2_float = Reduction.float(2.)

(* Création d'un objet "red_char_casse" de type Reduction avec la valeur de caractère
en majuscule 'A' *)
let red_char_casse = Reduction.char_casse('A');;

(* Création d'un objet "red_string" de type Reduction avec la valeur de chaîne de
caractères "abc" *)
let red_string = Reduction.string(Reduction.char) "abc";;

(* Création d'un objet "red_list" de type Reduction avec la valeur de liste de
caractères ['a'; 'b'; 'c'] *)
let red_list = Reduction.list(Reduction.char) ['a'; 'b'; 'c'];;

(* Création d'un objet "red_combine" de type Reduction en combinant deux objets
"Reduction.string(Reduction.char_casse)" et "Reduction.string(Reduction.char)" avec les
valeurs de chaînes de caractères "abc" et "def" *)
let red_combine = Reduction.combine(Reduction.string(Reduction.char_casse))
(Reduction.string(Reduction.char)) ("abc", "def");;
```

On effectue des réductions respectivement sur un flottant, un caractère, une string, une liste et enfin le couple de deux chaînes de caractères.

Pour l'exemple du caractère on utilise une fonction que l'on a ajouté qui applique une stratégie de réduction en fonction de la casse du caractère.

Enfin pour l'exemple de la liste et du couple on applique une réduction sur chaque élément de la liste ou du couple.

Module Test

```
#use "Property.ml" ;;
#use "Generator.ml" ;;
#use "Reduction.ml" ;;

module Test :
  sig
    (** Type d'un test portant sur des éléments de type 'a *)
    type 'a t

    (** Construit un test
      * @param gen  générateur pseudo-aléatoire de valeurs de test
      * @param red  stratégie de réduction
      * @param prop propriété qui fait l'objet du test
      * @return     test créé
      *)
    val make_test : 'a Generator.t -> 'a Reduction.t -> 'a Property.t -> 'a t

    (** Effectue un test
      * @param n    nombre de valeurs à tester
      * @param test test à effectuer
      * @return     `true` si n > 0 et que toutes les valeurs à tester satisfont les conditions
      *)
    val check : int -> 'a t -> bool

    (** Cherche une valeur simple ne vérifiant pas la propriété
      * @param n nombre de valeurs à tester
      * @return  `None` si toutes les valeurs de test générées par `gen` vérifient `prop`,
                 une valeur ne vérifiant pas `prop` (éventuellement en appliquant `red`) sinon
      *)
    val fails_at : int -> 'a t -> 'a option

    (** Exécute plusieurs tests
      * @param n     nombre de valeurs testées par test
      * @param tests liste des tests à vérifier
      * @return      tableau associatif des résultats
      *)
    val execute : int -> ('a t) list -> ('a t * 'a option) list
```

Description succincte de la solution mise en place

Le module Test permet de réaliser des tests sur une propriété à partir d'un générateur pseudo-aléatoire de valeurs et d'une stratégie de simplification des contre-exemples.

La fonction make_test crée un test à partir d'un générateur, d'une stratégie de réduction et d'une propriété. La fonction check permet de vérifier si toutes les valeurs générées par le test vérifient la propriété pour un nombre donné de tests. La fonction fails_at retourne la première valeur qui ne vérifie pas la propriété pour un nombre donné de tests. La fonction execute permet d'exécuter une liste de tests et de renvoyer les résultats sous forme de liste de paires (test, valeur échouée).

Difficultés rencontrées et leur résolution

L'inclusion des règles de Réduction dans le module Test a posé des difficultés lors de l'application de la stratégie de réduction sur une valeur générée avant la vérification de la satisfaction de la propriété.

Pour résoudre ce problème, nous avons simplifié la valeur générée en appliquant la stratégie de réduction de manière préventive avant de tester la propriété. Cela a garanti que la stratégie de réduction était appliquée sur toutes les valeurs générées sans affecter la vérification de la propriété.

Analyse de la pertinence des fonctionnalités, des avantages et des limites

L'avantage de cette solution est qu'elle permet de générer automatiquement des valeurs pour tester une propriété donnée, ce qui peut être très utile pour détecter des erreurs dans le code. Elle permet également de simplifier les contre-exemples pour faciliter le débogage. Cela peut permettre d'augmenter la couverture de test et donc la fiabilité du code.

Cependant, cette méthode ne garantit pas l'absence de bugs dans le code. Elle ne peut que détecter des erreurs dans les cas testés, qui peuvent ne pas couvrir toutes les situations possibles. En outre, la génération pseudo-aléatoire de données peut ne pas couvrir toutes les combinaisons possibles de valeurs, ce qui peut également limiter l'efficacité des tests.

Exemple d'application du module Test

Dans l'exemple donné, le module Test est utilisé pour tester la propriété de la fonction "abs_property" qui vérifie si une valeur est positive ou nulle. Un autre test est également effectué sur la même fonction avec une propriété différente. Ensuite, la fonction check est utilisée pour vérifier si toutes les valeurs générées par le test vérifient la propriété pour un nombre de tests donné, et la fonction fails_at est utilisée pour trouver la première valeur qui ne vérifie pas la propriété. Enfin, la fonction execute est utilisée pour exécuter une liste de tests et renvoyer les résultats sous forme de liste de paires (test, valeur échouée).

```
#use "Test.ml" ;;

(* Test Supplémentaire*)

let abs_property x = x >= 0.;;
let abs_property_bis x = x<0.;;
let abs_test = Test.make_test
    (Generator.float (-20.) 20.)
    (Reduction.float)
    (abs_property);;
let abs_test_bis = Test.make_test
    (Generator.float (-20.) 20.)
    (Reduction.float)
    (abs_property_bis);;
let result = Test.check 1000 abs_test;;
let fail= Test.fails_at 1000 abs_test;;
let execute = Test.execute 1000 [abs_test; abs_test_bis];;
Printf.printf "Test result: %b\n" result;;
```

exemples.ml

Description succincte de la solution mise en place

```
(* Test Supplémentaire*)
let abs_property x = x >= 0;;
let abs_property_wrong x = x<0;;
let abs_test = Test.make_test
    (Generator.int (-20) 20)
    (Reduction.int)
    (abs_property);;

let abs_test_wrong = Test.make_test
    (Generator.int (-20) 20)
    (Reduction.int)
    (abs_property_wrong);;

let result = Test.check 1000 abs_test;;
let fail= Test.fails_at 1000 abs_test;;
let execute = Test.execute 1000 [abs_test; abs_test_wrong];;

Printf.printf "Test result: %b\n" result;;
```

Le fichier exemple utilise concrètement les différents modules créés lors de ce projet et, en testant différentes choses, vérifie que les modules soient bien développés et fonctionnent bien les uns avec les autres. Plusieurs tests avaient déjà été réalisés dans le sujet du projet: test_quorem, test_quorem_wrong, test_append et test_append_wrong. Nous avons ajouté deux tests: abs_test et abs_test_wrong.

Difficultés rencontrées et leur résolution

Sur les tests déjà créés, au moment de la déclaration des tests, des strings sont placés entre le nom du test et les fonctions à tester. Cela empêche l'exécution des tests. Cependant, si l'on enlève ces phrases, les tests passent sans soucis. Peut-être est-ce une erreur lors de l'écriture du fichier.

D'autre part, lorsque l'on a créé nos tests, nous aurions aimé utiliser une fonctionnalité Generator.int_neg ainsi qu'un réducteur ne générant que des entiers négatifs. Nous n'avons pas pris le temps de coder de telles fonctionnalités et avons créé les tests actuels.

Analyse de la pertinence des fonctionnalités, des avantages et des limites

Dans ce fichier, différents tests sont réalisés, les tests test_quorem et quorem_wrong permettent de tester les générateurs et réducteurs d'entiers, les filtres sur les générateurs et réducteurs d'entiers ainsi que la combinaison de générateurs et celle de réducteurs. En outre, on utilisera les fonctionnalités check, fails_at et execute du module Test. Les tests test_append et test_append_wrong permettent de tester en plus, le générateur de listes ainsi que le réducteur de listes.
Les deux tests que l'on a créé, abs_test et abs_test_wrong permettent quant à eux de tester les générateur et réducteur de nombres décimaux.

Par contre, nous n'avons pas fait de tests sur les caractères et les chaînes de caractères.

Comme ce fichier n'est pas un module, on ne peut pas montrer d'exemples de son application.

Conclusion

Le projet d'API de property testing a été intégralement réalisé. Il s'exécute sans générer d'erreurs. Seul le code du fichier exemples.ml a été modifié afin de régler un problème présent dans le fichier original, en accord avec la professeur encadrante.

Des fonctionnalités et exemples supplémentaires ont été implémentés.

Dans sa globalité, le projet n'a pas été particulièrement difficile à réaliser. Cependant, nous avons eu du mal à comprendre ce qui était demandé. Il aurait été intéressant d'avoir un cas concret d'application de l'API.