RAPPORT DE PROJET OCAML:

Property Testing

Professeur encadrant : ROMUALD GRIGNON

Projet de groupe
avec la participation de :
BEWEKEDI Benjamin, SAGDULLIN Damir,
MOEUNG Alim, NGUYEN-CAO Quoc Thai (James),
TRAN Nhat Anh

31/03/2023

Sommaire :

1. INTRODUCTION

Le but principal de ce projet est de développer une libraire qui facilite l'écriture des tests en utilisant la vérification des propriétés grâce aux jeux de données générées aléatoirement.

De plus, pour faciliter le débogage, les jeux de données "plus simples" qui échouent au test sont proposés grâce aux méthodes de réduction.

Cela permet de détecter rapidement les erreurs dans les programmes, même pour des cas de test qui n'ont pas été spécifiquement envisagés lors de la conception du programme.

2. DESCRIPTION DE LA SOLUTION

Le code source du projet est réparti en 4 modules et un fichier avec des exemples d'utilisation de la librairie. Nous allons présenter brièvement chaque module ainsi que les fonctionnalités supplémentaires.

a) Module Property

Le module “Property” fournit des fonctions pour la gestion des propriétés booléennes sur des éléments de type ‘a. C'est la première étape de construction de tests : définition de propriétés à tester.

Le type "Property.t" est défini comme une fonction booléenne prenant en entrée un élément de type ‘a. Une propriété est considérée comme vraie si la fonction renvoie "true" pour cet élément.

Le module fournit également deux propriétés : "always_true" et "always_false" qui renvoient respectivement true et false à chaque appel. Elles sont utiles pour par exemple gérer les cas particuliers de certains tests.

La fonctionnalité supplémentaire proposée dans ce module permet de combiner facilement les “Property” grâce aux opérateurs logiques ET, OU, NON et IMPLIQUE. Sans cette interface les opérateurs fournis par OcaML “&&”, “||” et “not” ne sont pas utilisables avec “Property”.

b) Module Generator

Le module “Generator” permet de générer et de gérer les jeux de données pseudo-aléatoires sur lesquelles on va tester les propriétés définies grâce au module “Property”.

La partie de son interface qui s’occupe de la génération des données est composée de :

- const : cette fonction prend en paramètre une valeur x et retourne un générateur dont l’unique valeur sera x.

- bool : cette fonction prend en paramètre un float prob qui est la probabilité de la valeur ‘true’ , dans cette fonction on utilise Random.float permet de choisir un float aléatoirement entre 0 et 1.Si la valeur généré est inférieur à prob,la fonction retourne ‘true’, sinon ‘false’.

- int : cette fonction prend en paramètre deux entiers,inf pour la borne inférieur et sup pour la borne supérieur, cette fonction utilise Random.int qui permet de générer un entier aléatoirement compris dans l’intervalle [inf,sup].

- int_nonneg : cette fonction prend en paramètre un entier n,cette fonction utilise Random.int qui génére aléatoirement un entier dans l’intervalle [0,n].

- float : cette fonction prend en paramètre 2 float inf pour la borne inférieure et sup pour la borne supérieure, la fonction utilise Random.float pour générer un float compris dans l’intervalle [inf,sup].

- float_nonneg : cette fonction prend en paramètre un float n, la fonction utilise Random.float pour générer aléatoirement un float dans l’intervalle [0,n].

- char : cette fonction définit une donnée rand_code qui est entier généré aléatoirement dans l’intervalle [0,52].On génère un caractère avec Char.chr qui prend en paramètre le code ASCII,si rand_code est supérieur à 25, on ajoute 39 à rand_code pour obtenir un caractère dans l’intervalle [A-Z],sinon on ajoute 97 à rand_code pour obtenir un caractère dans l’intervalle [a-z].

- alphanum : cette fonction est similaire à la précédente : il y a une condition supplémentaire, si rand_code est un entier supérieur à 51 alors on soustrait 4 à rand_code pour générer un caractère alphanumérique dans l’intervalle [0-9].

- string : cette fonction prend en paramètre la taille de la chaîne de caractère que l’on souhaite générer et gen pour le type de générateur gen,dans cette fonction

nous avons défini une fonction récursive aux qui prend en paramètre un indice i et la liste que l'on construit acc, si l'indice i est égale à la taille n, on retourne la liste inversée de acc que l'on a construite, sinon on génère un élément x avec le générateur gen, ensuite on ajoute x à l'indice i et on appelle la fonction aux avec l'indice i + 1 et la chaîne de caractère que l'on construit au fur et à mesure.

Ensuite, l'interface qui permet de manipuler les générateurs est constituée de :

- next : cette fonction prend en paramètre un générateur pseudo aléatoire, la fonction retourne une nouvelle valeure généré le dernier.

- combine : prend en paramètre 2 générateur fst_gen et snd_gen , et retourne un couple de 2 valeurs généré par ces 2 générateurs.

- map : cette fonction prend en paramètre une fonction p et un générateur gen , map retourne un générateur en appliquant la fonction f sur celui-ci.

- filter : cette fonction prend en paramètre une condition p et un générateur gen, on génère une élement x avec gen, si x vérifie la condition p, alors on retourne x , sinon on régénère un nouvel élément x (on fait ça tant que x ne satisfait pas la condition p,d'où la récursivité)
- partitionnned_map :cette fonction prend en paramètre , elle retourne un générateur fst f' pour toute valeur vérifiant `p` et `snd f` pour toute valeur ne le vérifient pas.

En ce qui concerne les fonctionnalités en plus, chaque générateur de données à été couplé à un générateur précédé de "_seed" prenant en paramètre en plus : une graine qui permet la génération des mêmes valeurs pseudo-aléatoires. Cela simplifie le débogage car la génération devient déterministe et donc il est plus facile de reproduire les erreurs.

c) Module Reduction

Dans le cas où on rencontre une valeur qui ne vérifie pas la propriété testée, on va essayer d'en déduire un contre-exemple "plus simple". Que signifie "plus simple" et quelles sont les stratégies que l'on a adoptées, les réponses se trouvent dans la description des fonctions qui suit.

Le type "Reduction.t" est défini comme une fonction prenant en entrée une valeur de type abstrait et retournant une liste de valeurs de ce type.

- empty : stratégie vide qui retourne la liste vide peut importe l'élément à réduire

- int : stratégie de réduction des entiers, par exemple pour l'entier n, cela retourne [0;1;-1;2;-3;....;n;-n] c'est à dire tous les entiers entre -n et n inclusen considérant comme plus simple l'entier ayant la valeur absolue la plus petite, en cas d'égalité, l'entier positif est considéré plus simple que son homologue négatif.

- int_nonneg : stratégie de réduction sur les entiers positifs, cela retourne tous les entiers entre 0 et n exclus, par exemple pour l'entier n, cela retourne [0;1;2;3;4;5;...;n] en considérant comme plus simple l'entier le plus petit.

- float : stratégie de réduction des flottants par exemple pour le flottant n, cela retourne un ensemble de flottant entre -n et n inclus avec un pas de 0.5, en considérant comme plus simple le flottant ayant la valeur absolue la plus petite, en cas d'égalité, le flottant positif est considéré plus simple que son homologue négatif.

- float_nonneg : stratégie de réduction sur les flottants positifs, cela retourne un ensemble de flottants entre 0 et n, avec un pas de 0.5, en considérant comme plus simple le flottant le plus petit.

- char : stratégie de réduction sur les caractères alphabétiques (majuscules A-Z et minuscules a-z). Cela retourne l'ensemble des caractères inférieurs au caractère d'entrée selon leur code ASCII respective. Ainsi, est considéré comme plus simple le caractère ayant le code ASCII le plus petit.

- alphanum : stratégie de réduction pour les caractères alphanumériques (A-Z a-z 0-9).Cela retourne l'ensemble des caractères inférieurs au caractère d'entrée selon leur code ASCII respective. Ainsi, est considéré comme plus simple le caractère ayant le code ASCII le plus petit.

- string : stratégie de réduction pour les chaînes de caractères. Cela prend en entrée une stratégie de réduction pour les caractères et un string et renvoie une

liste de chaînes de caractères obtenus par la concentration des réductions des caractères composant notre chaîne de caractères en entrée. Est considéré comme plus simple la concaténation des caractères les plus simples obtenus à partir de la réduction des caractères de notre chaîne de caractère d'entrée.

- carac_par_carac est une fonction auxiliaire qu'on utilise pour Reduction.string. Le but de cette fonction est de passer d'un string à une liste de caractère. Par exemple "Hello" => ['H';'e';'l';'l';'o']
- ligneMot est aussi une fonction auxiliaire qu'on utilise pour Reduction.string. Cette fonction permet de générer un string à par d'un char list list correspondant aux réductions des caractères de notre string d'entrée.
  exemple :

i	H	e	l	l	o
0	A	A	A	A	A
1	B	B	B	B	B
….					
30	e	e	e	e	e
31	f		f	f	f
…					

- list : stratégie de réduction sur les listes. Cela prend en entrée une stratégie de réduction et une liste d'éléments correspondant à cette stratégie de réduction et retourne une liste de liste où chaque "sous liste" i est composé d'un élément aléatoire parmi la liste de réduction des éléments de la liste à réduire.
- newList : c'est une fonction auxiliaire qu'on utilise pour Reduction.list. Le but de cette fonction est de générer une nouvelle liste à partir de la réduction des éléments de la liste d'entrée. En effet, pour chaque élément, elle le remplace par un élement de sa réduction pris au hasard.

- combine : stratégie de réduction pour les couples. Cela prend en entrée une stratégie de réduction correspondant au membre de gauche et une stratégie de réduction correspondant au membre de droite (et donc un couple compatible aux stratégies de réductions). Ensuite, cette stratégie de réduction retourne une liste dont le nombre d'éléments est égal au max du nombre d'éléments de la réduction des deux éléments couples. Une fois que la liste la plus petite est finie, on pioche à nouveau sur le premier élément de cette liste (utilisation du modulo). Est considéré comme plus simple le couple comportant les réductions les plus simples des deux éléments du couple d'entrée.

d) Module Test

Le module "Test" fournit des fonctions pour mettre en relation les trois autres modules. Il comprend des fonctions pour créer un test, effectuer un test, chercher une valeur “plus simple” qui ne satisfait pas la propriété et exécuter plusieurs tests.

Le type "Test.t" est défini comme un enregistrement contenant un générateur de valeurs, une stratégie de réduction, un nom de test et une propriété à tester.

- make_test : permet de créer un test .

- check permet de vérifier si toutes les valeurs générées par le générateur de valeurs vérifient la propriété à tester. Elle prend en entrée le nombre de valeurs à tester et le test à effectuer.

- fails_at : permet de trouver une valeur “plus simple” qui ne satisfait pas la propriété à tester après avoir appliqué une stratégie de réduction. Elle prend en entrée le nombre de valeurs à tester et le test à effectuer. Si toutes les valeurs générées par le générateur de valeurs vérifient la propriété, la fonction renvoie "None".

- execute : permet d'exécuter plusieurs tests en une seule fois. Elle prend en entrée le nombre de valeurs à tester pour chaque test et une liste de tests à effectuer. Elle renvoie un tableau associatif contenant les résultats pour chaque test, sous forme d'une paire du test et de retour de fails_at.

En plus de ces fonctionnalités basiques on a ajouté deux fonctions :

- checkPercentage : permet d’avoir le pourcentage de valeurs générées qui vérifient la propriété et le pourcentage de valeur générée qui ne la vérifie pas (sur n valeurs).

- fails_at_init : identique à failt_at mais retournant une couple de valeur en cas de non validation du test de forme : (valeur initiale causant l’échec, valeur réduite). Elle met en évidence l’intérêt des méthodes de réduction.

e) Exemples d'utilisation

Le fichier "exemples.ml" implémente un ensemble d'exemples de tests. Les tests portent sur les fonctions de base telles que la division euclidienne, la concaténation de listes, l'inversion de listes, l'application d'une fonction à chaque élément d'une liste, le filtrage des éléments d'une liste, le tri d'une liste et le produit scalaire.

Pour utiliser la bibliothèque il faut :

1. Définir des générateurs (éventuellement avec une graine partagé par un autre développeur qui a signalé un bug) et des réducteurs pour les types de données sur lesquels les tests seront effectués.

2. Construire le test pour chaque propriété en utilisant "make_test" qui prend en argument un générateur, un réducteur, le nom et une propriété à tester.

3. Enfin, il suffit d'appeler "check" pour exécuter un test, "fails_at" si on souhaite voir une entrée simple qui fait échouer notre code ou "execute" pour exécuter plusieurs test. Accessoirement, on a la possibilité de consulter la proportion des propriétés non vérifier sur l'ensemble du jeu de données généré avec "checkPercentage" ou de trouver la valeur initiale qui a causé l'échec avec "fails_at_init".

Ainsi, lorsqu'on code par exemple une fonction de trie, on peut utiliser cette bibliothèque afin de vérifier si la fonction trie effectivement une liste, dans le cas contraire, on obtient un contre exemple qui peut nous aider à déboguer notre fonction de trie.

3. ANALYSE DU PROJET

a) Avantages

Pertinence de la solution finale

- typage statique et vérification robuste des signatures: comme OCaml étant un langage à typage statique avec inférence de type, les développeurs peuvent améliorer leurs compétences en écriture de code robuste et fiable, ce qui est bénéfique pour tous les projets sur lesquels ils travaillent.

- facile à utiliser pour propriétés complexes car composition des fonctions permet d'empiler les couches graduellement: La programmation fonctionnelle favorise la composition de fonctions pour construire des programmes complexes à partir de fonctions simples. En Ocaml, on peut facilement définir des fonctions qui prennent en entrée des fonctions et qui retournent des fonctions, ce qui permet de composer des fonctions de manière modulaire et d'éviter la duplication de code.

- facile à ajouter de nouvelles fonctionnalités : sa flexibilité et à la possibilité de définir des modules et des fonctions réutilisables.

- optimisation des fonctions "tail-recursive" par compilateur: Le compilateur peut transformer la fonction récursive en une boucle itérative, ce qui permet d'éviter l'accumulation de la pile d'appels récursifs. L'optimisation des fonctions "tail-recursive" est effectuée automatiquement par le compilateur d'Ocaml, sans que le développeur ait besoin de prendre des mesures spéciales pour cela. Cela permet aux développeurs de se concentrer sur la logique de leur code plutôt que sur les détails de l'optimisation de la performance.

- outil assez puissant avec une architecture simple : grâce à sa bibliothèque standard riche et ses fonctionnalités avancées telles que le typage fort, l'inférence de type, la gestion de mémoire automatisée, les closures, la généricité.

b) Limites

- Compliqué de changer les signatures des fonctions car utilisées dans beaucoup de contextes car cela peut avoir des répercussions sur d'autres parties du code qui utilisent ces fonctions.

- Manque de documentation : bien que l'écosystème OCaml soit en croissance, il reste moins développé que celui de langages plus populaires. Il peut donc être plus difficile de trouver des tutoriels, des exemples de code ou des bibliothèques pour certaines fonctionnalités spécifiques.

c) Répartition des tâches/difficultés rencontrées

Le fait que le langage OCamL est fortement typé nous a permis de partager facilement les tâches et de les faire en parallèle.

Ci dessous vous trouverez la répartition approximative des tâches car chacun a participé à un minima dans tous les aspects du projet :

Property	Damir
Generator	Benjamin, Alim, James, Damir
Reduction	Alim, Benjamin
Test	Damir, Alim
Exemples	Damir, James, Nhat-Anh, Alim
Rapport	Ensemble

Notre première difficulté était bien évidemment la compréhension globale du sujet, son intérêt et des liens entre les quatre modules.

Ensuite, notre plus grande difficulté du projet était dans la définition de “plus simple” pour nos stratégies de réduction. En effet, ce n’était pas évident de définir la notion de simplicité d’autant plus qu’on a commencé par coder Réduction et Generator.

L’autre difficulté qui est d’ailleurs la résultante de celle-ci est qu’il était donc tout aussi difficile de trouver un consensus sur cette notion de simplicité. Ainsi, si pour des classes comme int/float/etc. le consensus était plus facile, cela a été plus délicat pour le cas des réductions de string ou list.

Enfin, notre dernière difficulté a été de trouver des exemples cohérents et pas trop banales. Tout comme l’ajout de fonctionnalités, la difficulté était de ne pas trop ajouter des fonctionnalités qui nécessitent une refonte totale de nos fonctions.

OCaml est un langage de programmation fonctionnel et qu’on a pas l’habitude d’utiliser ce type de paradigme, il nous a donc fallu un petit temps d'apprentissage et d’adaptation afin de se familiariser avec la syntaxe et les concepts de programmation fonctionnelle.

4. CONCLUSION

Ce projet en Ocaml a atteint son but académique, permettant aux membres de l'équipe de mieux comprendre et d'avancer dans le paradigme fonctionnel qui n’a pas été simple pour nous au début. Les fonctionnalités de base ont été réussies, mais pour créer une librairie fonctionnelle et réutilisable, il faudra encore beaucoup de travail. En parallèle, cela nous a rappelé l’important de travailler davantage sur la gestion du git et la répartition des tâches afin de faciliter le travail en équipe. 👍🎉