Infrastructure de tests

Projet Programmation Fonctionnelle 2SN - Systèmes Logiciels - Session 1 - 2020/2021

1 Sujet d'étude

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.

1.1 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. Ceci correspond à la sémantique d'une précondition, i.e. on n'exécute un programme que si sa précondition est vraie, i.e. on ignore les autres exécutions.
- 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. Ceci correspond à la sémantique d'une post-condition, i.e. on ne valide l'exécution que si la postcondition est vraie, i.e. les autres exécutions sont invalides et correspondent à une erreur.
- miracle : unit -> 'a, qui interrompt l'exécution et la rend valide (sans renvoyer de valeur).
- failure : unit -> 'a, qui interrompt l'exécution et la rend invalide (sans renvoyer de valeur).
- 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 ssi les deux exécutions "filles" le sont.
- forsome_bool : unit -> bool, qui "forke" de la même façon l'exécution courante. L'exécution "parente" est valide ssi au moins une exécution "fille" l'est.
- forall : 'a Flux.t -> 'a, qui généralise forall_bool et "forke" l'exécution courante en autant de versions qu'il y a d'éléments dans le flux.
- forsome : 'a Flux.t -> 'a, qui "forke" de la même façon l'exécution courante et généralise forsome_bool.
- foratleast : int -> 'a Flux.t -> 'a, qui "forke" de la même façon l'exécution courante. L'exécution "parente" est valide ssi au moins n exécutions "filles" le sont. On a forsome = foratleast 1.
- 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`.

1.2 Exemples

• On peut instrumenter de la façon suivante le programme premier, qui détermine si un entier est premier, pour tester que pour tout entier non premier a entre 2 et 50, il existe b≠a tel que : a%b=0.

```
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))
```

• On peut tester que pgcd vérifie $pgcd(a, b) = 1$ pour tous entiers a et b entre 1 et 20 premiers entre eux [1].

```
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))
```

1.3 Choix d'implantation

Le non-déterminisme, dans un cadre purement fonctionnel, peut s'implanter de plusieurs façons. Une solution monadique est possible, mais elle oblige à instrumenter et modifier tous les programmes que l'on désire tester par ce moyen. Les sources des programmes n'étant pas toujours disponibles et leur modification avant de les tester pas souhaitable, nous n'envisagerons pas cette solution.
Un second choix d'implantation consiste à utiliser les continuations et les possibilités qu'elles offrent en terme de modification du contrôle, afin d'implanter le non-déterminisme. Le principe est le suivant : lorsqu'on veut "forker" une exécution, on commence par capturer ce qu'il reste à exécuter dans une continuation, puis à relancer directement cette dernière autant de fois que souhaité avec des valeurs de reprise différentes prises dans le flux passé en paramètre. Il est également possible de récupérer le résultat de tous ces calculs pour savoir si une exécution "fille" est valide ou non. Toutes ces captures de continuation sont encapsulées dans un unique ' reset ' défini par la fonction check.
Le type (de retour) des exécutions doit simplement distinguer les deux cas "valide" et "invalide". Dans un premier temps, il n'y a pas d'autre information/donnée supplémentaire à représenter, mais cela peut changer selon les extensions considérées (cf. section 2).
Enfin, il est fortement conseillé d'implanter forall , forsome et foratleast en utilisant forall_bool , miracle, forsome_bool et failure , sans avoir recours directement à Delimcc.push_prompt et Delimcc. shift .

2 Extensions

Ces extensions ne sont à envisager qu'après avoir correctement implanté les fonctionnalités précédentes.

2.1 Quantificateurs sur les listes

On souhaite également avoir des quantificateurs permettant de traiter une spécification de la forme : "quelle que soit la longueur d'une liste l, il existe un élément de l tel que . . .". De tels quantificateurs sur la **structure/forme** de la liste (i.e. sa longueur) sont nécessairements suivi d'un autre sur les éléments. On leur donne

1. On suppose défini le prédicat premiers_entre_eux qui teste si deux entiers sont bien premiers entre eux.

le type suivant, où le paramètre de type `int Flux.t` est le flux des longueurs possibles sur lesquelles on quantifie, le paramètre de type `unit -> 'a` est le quantificateur sur les éléments de la liste, le même quantificateur pour tous :

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

Par exemple, on pourrait ainsi spécifier la fonction `mem` (du module `List`), en se basant sur la propriété $mem\ x\ l = \exists l1, l2.l = l1@(x :: l2)$:

```
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 > 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)))
  )
```

On pourrait ici spécifier plus finement les longueurs possibles des listes `l1` et `l2`, en fonction de celle de `l`.
Note : pour reprendre l'exemple précédent, il y a une différence fondamentale entre :

```
let e = forsome values in
let l1 = forsome_length lengths (fun () -> e) in
...
```

et :

```
let l1 = forsome_length lengths (fun () -> forsome values) in
...
```

Dans le premier cas, on choisit un entier `e` dans `values` et on construit une liste d'une longueur choisie dans `lengths` dont tous les éléments sont égaux à `e`. Dans le second cas, pour chaque élément de la liste à choisir, on appelle `(fun () -> forsome values)` qui effectue un choix indépendant des autres.

2.2 Logger les exécutions valides / invalides

On se propose d'ajouter les primitives suivantes, qui permettent de débugger/logger les exécutions pour savoir si elles sont valides ou non et de mieux comprendre ce qui s'exécute.

- `on_success : (unit -> unit) -> unit` qui évalue le programme passé en paramètre ssi l'exécution courante est valide.
- `on_failure : (unit -> unit) -> unit` qui évalue le programme passé en paramètre ssi l'exécution courante est invalide.

Note : les programmes passés en paramètre aux primitives ci-dessus sont bien évalués dans l'ordre de leur apparition dans un programme. En revanche, ils ne peuvent être évalués qu'à la fin d'une exécution, par la fonction `check`, lorsqu'on sait si elle est valide ou non [2]. Le pricipe de `on_success` et `on_failure` est d'accumuler ces programmes de débuggage, en plus du caractère valide ou invalide de l'exécution.

2. Ces programmes ne peuvent donc influencer l'exécution courante et changer son résultat par exemple

À titre d'exemple, le programme instrumenté suivant permet de tester les triplets d'entiers x, y et z tels que $x = 5y + 7z$. On cherche plus particulièrement à déterminer les x tels que la propriété $P(x) = \exists y, z \in [0, 20].x = 5y + 7z$ est vraie ou non, et à savoir finalement si $Q = \forall x \in [10, 50].x\%2 = 0 \rightarrow P(x)$ est valide ou non, en affichant les éventuels contre-exemples :

```
(* le prédicat de base à tester *)
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
    (* affiche la validité (ou non) de la propriété Q *)
    on_success (fun () -> Format.printf "Q est valide@.");
    on_failure (fun () -> Format.printf "Q est invalide@.");
    let x = forall (interval 10 50) in
    (* affiche les contre-exemples de la propriété P(x) *)
    on_failure (fun () -> Format.printf "non P(%d)@." x);
    (* on ne s'intéresse qu'aux entiers x pairs *)
    assumption (fun () -> x mod 2 = 0);
    let y = forsome (interval 0 20) in
    let z = forsome (interval 0 20) in
    (* affiche tous les triplets x, y, z vérifiant le prédicat de base *)
    on_success (fun () -> Format.printf "%d = 5 * %d + 7 * %d@." x y z);
    assertion (fun () -> pred x y z)
  end);;
```

Dans cet exemple, la propriété Q est invalide car $P(16)$ est faux. En supposant que les entiers sont traités dans l'ordre croissant, on aura les affichages :

```
Q est invalide
10 = 5 * 2 + 7 * 0
12 = 5 * 1 + 7 * 1
14 = 5 * 0 + 7 * 2
non P(16)
```

Vous devez donc proposer un mécanisme pour collecter les programmes paramètres de on_success et on_failure au fur et à mesure d'une exécution et n'exécuter que les programmes correspondant à une exécution valide ou invalide, à la fin de celle-ci, selon sa validité. L'ordre d'affichage n'est cependant pas unique et dépend de l'ordre dans lequel vous traitez les différentes valeurs possibles produites par un quantificateur.

3 Raffinement de programmes

On peut utiliser les primitives proposées pour définir des exemples plus sophistiqués et se rapprocher de ce que l'on appelle le développement par "raffinement de programmes". Par exemple, on peut vouloir s'assurer que l'algorithme du *merge sort* (vu en TP 2 sous le nom "tri fusion") fonctionne quelque soit la façon dont la fonction split : 'a list -> 'a list * 'a list coupe la liste en deux. On peut ainsi spécifier la fonction split , même si on ne l'a pas définie, et vérifier que mergesort fonctionne dans tous les cas. Pour cela, on utilise la structure monadique des flux et la version monadique de la fonction permutations (transposée à Flux) définie en TD/TP :

```
(* recomposition (voir TP 2) *)
let rec merge l1 l2 = ...
(* décomposition par programmation non-déterministe. *)
(* split l renvoie toutes les décompositions possibles de l dans l1 et l2 *)
let split l =
  let len = List.length l in
  (* pour couper une liste en 2 à une position donnée *)
  let rec split_after l n =
    if n = 0 then ([], l) else let (l1, l2) = split_after (List.tl l) (n-1) in ((List.hd l)::l1, l2) in
  (* les valeurs de l1 et l2 sont à choisir dans une permutation de l *)
  H.forall Flux.(permutations l >>= fun perm ->
                 let (l1, l2) = split_after perm (len/2) in
                 (* 2 longueurs possibles pour l1 et l2: len/2 et len/2 + 1 *)
                 Flux.(return (l1, l2) ++ (if len mod 2 = 0 then zero else return (l2, l1))))
(* tri fusion récursif classique *)
let rec mergesort l =
  match l with
  | [] | [_] -> l
  | _        -> let (l1, l2) = split l in merge (mergesort l1) (mergesort l2)
(* test du tri fusion pour tous les splits possibles *)
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
  (* on compare mergesort avec List.sort *)
  assertion (fun () -> List.sort Pervasives.compare l = mergesort l))
```

4 Critères de jugement

- dans tous les cas, votre application doit s'exécuter.
- la lisibilité du code et la qualité des contrats et des commentaires sera évaluée.
- le code devra être **purement fonctionnel**.
- il est demandé un effort de **conception**, en utilisant des modules, des interfaces de module, des foncteurs de module, etc.
- les fonctionnalités demandées doivent former une librairie indépendante qu'on peut utiliser pour tester n'importe quelle autre application.
- vous devez choisir et implanter 1 **extension** au minimum parmi les 2 proposées. Vous veillerez à bien indiquer dans votre code et rapport les extensions choisies.
- les fonctionnalités minimum attendues ainsi que les extensions choisies devront être validées à travers des exemples concrets de spécifications et de tests.
- l'effort avec lequel vous avez intégré des extensions, au delà du minimum requis, sera apprécié.

5 Travail et Rendu

- ce projet sera effectué par groupes de 3 ou 4 personnes.
- à l'issue de celui-ci, une archive contenant vos fichiers sources, vos tests ainsi qu'un rapport sera remise sous Moodle.
- le projet débute le vendredi 11 décembre 2020 et s'achève le vendredi 22 janvier 2020 à 18h.