Programmation en OCamL 4
Matthieu Boyer
Planche A

Khôlle MP2I
04/11/2024

Famille de Formes et Fonction d'Ordre Supérieur

On se donne le type OCamL suivant pour représenter les formes paramétrées dans l'espace. On ne conservera pas d'information de position dans la structure.

```
type forme = {dimensions: float * int; volume: float -> float};
```

En deux dimensions, le volume, c'est l'aire. Toutefois, pour un objet qui peut se représenter dans le plan, si on le représente dans l'espace en 3-dimensions, son volume devient nul, d'où le paramètre entier.

1. Donner la représentation du cercle unité dans le type `float forme`.
2. Donner la représentation de la boule unité dans le type `float forme`.
3. Donner une fonction `vol: forme -> float` renvoyant le volume d'une forme donnée.
4. Donner une fonction `flatten: forme list -> 'a list forme` qui a une collection d'objets renvoie l'objet représentant la collection. On ne s'intéressera qu'aux objets qui, plongés dans un même espace sont de volume non nul, c'est à dire, aux objets dont la dimension est maximale.

Fenêtre sur les Foncteurs et Monades

Cet exercice était déjà présent dans la feuille précédente, mais est en rapport avec l'exercice 1 Voici un type paramétré OCamL : `type 'a maybe = None | Some of 'a;;`

1. Donner un exemple de programme pour lequel le type `int maybe` pourrait être utile.

On va s'intéresser aux manières de faire des transformations sur des objets de type `'a maybe`.

2. Définir une fonction `map: ('a -> 'b) -> 'a maybe -> 'b maybe = <fun>` vérifiant les deux propriétés suivantes :
   (a) `map (fun x -> x) f = f;;`
   (b) `map foo (map bar f) = map (fun x -> foo (bar x)) f;;`

On appelle un type paramétré `f` muni d'une telle fonction `map` un *foncteur*[a]. `maybe` est un exemple de foncteur.

3. Définir le foncteur `lst` dont la fonction `map` est l'application point à point.

Une propriété importante des foncteurs est que la composition de deux foncteurs doit être un foncteur : étant donné un foncteur `'a foo` et un foncteur `'a bar`, on peut définir un foncteur `'a foobar = ('a bar) foo` et définir une fonction `mapFooBar`.

4. Définir le foncteur composition de `maybe` et de `lst`.

On définit de manière similaire la notion de monade. Une monade est un type paramétré `'a f` muni d'une fonction `fmap: ('a -> 'b f) -> 'a f -> 'b f`.

5. Donner les fonctions `fmap` pour les foncteurs précédents.

a. En réalité c'est plus compliqué que cela mais bon...

Programmation en OCamL 4
Matthieu Boyer
Planche A : Corrigés

Khôlle MP2I
04/11/2024

Solution: Famille de Formes et Fonction d'Ordre Supérieur

1. Par exemple :

```
let v f = 3.14 *. f *. f;; let cercle = {dimensions= 1. * 2; volume= v}
```

2. Par exemple :

```
let v f = 1.33*. 3.14 *. f *. f *. f;; let cercle = {dimensions= 1. * 3; volume= v}
```

3. Tout simplement `let vol f = f.volume (fst f.dimensions);;`
4. On va définir deux fonctions :

```
let get_dim formeliste =
let rec aux ok_dim =          | [] -> []
        | h::t -> if (snd h.dimensions) < ok_dim then aux ok_dim t
                  else if (snd h.dimensions) > ok_dim then h::(aux (snd h.dimensions) t)
else h::(aux ok_dim t)
;;

let v formeliste dimensionliste =
        let aux acc vi di = acc +. (vi di) in
        List.fold_left2 aux (List.map (fun x -> x.volume) formeliste) dimensionliste;;

let flatten formeliste =
        let formeliste_ok = get_dim formeliste in
        let dimensionliste = List.map (fun w -> w.dimensions) formeliste_ok in
        {dimensions = ((fst List.head dimensionliste), snd List.head dimensionliste);
                volume = v formeliste_ok};;
```

Solution: Fenêtre sur les Foncteurs et Monades

1. Pour la fonction `div: int -> int -> int maybe`.
2. `let maybemap f = function None -> None | Some x -> Some (f x);;`
3. Comme en cours :

```
type 'a lst = Nil | Cons of 'a * 'a lst;;

let rec lstmap f = function Nil -> Nil | Cons(a, l) -> Cons (f a) (map l);;
```

4. Comme écrit au dessus :

```
type 'a maybelst = None | Some of 'a lst;;

let maybelstmap f ml = maybemap (lstmap f) ml;;
```

5. On donne :

```
let maybefmap f = function
        | None -> None
        | Some x -> f x
;;

let rec concat l1 l2 = match l1 with
        | [] -> l2
        | Cons(x, l') -> Cons(x, (concat l' l2))
;;

let rec lstfmap f = function
        | Nil -> Nil
        | Cons(x, l) -> concat (f x) (lstfmap f l)
;;

let maybelstfmap f ml = maybefmap (lstfmap f) m:w l;;
```

Programmation en OCamL 4
Matthieu Boyer
Planche B

Khôlle MP2I
04/11/2024

Interpréteur de Fonction et Dérivation

On définit les types suivants :

```
type unop = Moins | Log ;;
type binop = Plus | Moins | Fois | Exp;;
type fonction = Var | Const of float
        | UOp of unop * fonction
        | BOp of binop * fonction * fonction;;
```

1. Décrivez en OCamL les fonctions $id : x \mapsto x$, $double : x \mapsto 2x$ et $sqr : x \mapsto x^2$.
2. Donner une fonction `interprete: fonction -> float -> float` qui renvoie la valeur d'une fonction dans un environnement.
3. Donner une fonction `deriv: fonction -> fonction` qui renvoie l'expression de la dérivée d'une fonction. On supposera que le deuxième argument de l'exposant est une constante.
4. Bonus : Quid de la question de la primitivation ? Et de l'intégration ?

Programmation en OCamL 4
Matthieu Boyer
Planche B : Corrigés

Khôlle MP2I
04/11/2024

Solution: Interpréteur de Fonction et Dérivation

Pour les deux questions de cet exercice, on procède simplement par induction.

Programmation en OCamL 4
Matthieu Boyer
Planche C

Khôlle MP2I
04/11/2024

Mini Typeur

On se donne les types suivants pour la représentation d'un langage de programmation MiniML :

```
type types = Int | Bool ;;
type unop = Moins | Non;;
type binop = Plus | Fois | Et | Ou | Eq | Ge | Gt;;
type program = IConst of int
        | BConst of bool
        | Uop of unop * program
        | BOp of binop * program * program
        | IfElse of program * program * program;;
```

1. Donnez le programme représentant les opérations équivalentes suivantes (on ne vérifiera pas les types) :
   - $2 + 2$
   - $true \wedge 3$
   - $\begin{cases} 1+1 & \text{si } 3 = 4 \\ 2+2 & \text{sinon} \end{cases}$

On ne peut additionner et multiplier que des entiers, ce qui renvoie des entiers. On peut comparer tous éléments, ce qui renvoie des booléens. On ne peut prendre la conjonction et la disjonction que de booléens, ce qui renvoie un booléen. L'opérateur `Moins` prend un entier et renvoie un entier, l'opérateur `Non` prend un booléen et renvoie un booléen. La construction `IfElse` prend un programme de type booléen et deux programmes de même type t et a donc pour type t.

1. Donnez une fonction `typeur: program -> types` qui renvoie le type associé à un programme. On ne vérifiera pas à ce stade si le programme est bien typé.
2. Donnez une fonction `bien_typé: program -> bool` qui renvoie `true` si et seulement si le programme est bien typé.

Programmation en OCamL 4
Matthieu Boyer
Planche C : Corrigés

Khôlle MP2I
04/11/2024

Solution: Mini Typeur

Le typeur n'est pas très complexe en procédant par induction. Pour la vérification du correct typage, il faut penser à appeler récursivement le typeur sur les sous-programmes pour vérifier si le sous-programme est bien du bon type *et* vérifier que le sous-programme est bien typé.