Institut National Universitaire Champollion

L3 Info
Algorithmes, types de données et preuves

Semestre 2
Année 2024 - 2025

TP 1: Typage de programmes fonctionnels

1 Fonctions d'ordre supérieur

Écrivez les fonctions d'ordre supérieur suivantes, voir aussi https://ocaml.org/manual/5.3/api/List.html#1_Listscanning

- `map`
- `for_all`
- `exists`
- `filter`
- `partition`

2 Représentation de Caml en Caml

Dans ce TP, nous commençons à implanter une partie de Caml en Caml:

- vérification de types
- inférence de types
- moteur d'évaluation

Ici, nous nous intéressons à représenter un langage fonctionnel de base en Caml et de vérifier son typage (§ 4). Vous pouvez ensuite passer à un langage plus étendu (§ 5).

La matière est difficile à comprendre à cause de l'auto-référentialité de ce que nous faisons: nous utilisons Caml en même temps comme *langage de programmation* dans lequel nous écrivons des fonctions, et comme *langage à représenter*.

Exercice 1 (Représentation de types) Nous proposons de représenter les types à l'aide des types inductifs suivants (comparez avec les transparents du cours, p. 25):

```
type base_tp =
  BoolT
| IntT

type tp =
  ConstT of base_tp
| FunT of tp * tp
```

Par exemple, le type Caml `int -> bool` sera représenté par le terme du type `tp` suivant: `FunT (ConstT IntT, ConstT BoolT)`.

Représentez les types suivants:

1. `int -> (bool -> int)`
2. `(int -> bool) -> int`
3. `(int -> int) -> (int -> int)`

Exercice 2 (Représentation d'expressions) Nous proposons de représenter les expressions à l'aide des types inductifs suivants (comparez avec les transparents du cours, p. 26):

```
type const_expr =
  BoolE of bool
| IntE of int

type expr =
    Const of const_expr
  | Var of string
  | Abs of string * tp * expr
  | App of expr * expr
```

Par exemple, l'expression Caml `(fun (x : int) -> x) 2` sera représentée par le terme du type `expr` suivant: `App (Abs ("x", ConstT IntT, Var "x"), Const (IntE 2))`.

Représentez les expressions suivants:

1. `((fun (x : int) -> x) 2) 3`
2. `(fun (x : int) -> fun (y : bool) -> y) 2 3`
3. `(fun (f: int -> bool) -> f 2) (fun (x: int) -> true)`

Vous remarquez que l'on peut représenter des expressions Caml qui ne sont pas bien typés.

Exercice 3 (Affichage de types) Écrivez la fonction `string_of_type` qui affiche un type, étant donnée sa représentation interne. Cette fonction est donc l'inverse de la traduction de l'exercice 1.

Exercice 4 (Affichage d'expressions) Écrivez la fonction `string_of_expr` qui affiche une expression, étant donnée sa représentation interne. Cette fonction est donc l'inverse de la traduction de l'exercice 2.

3 Interlude: Listes d' association et maps

Le type

```
type 'a option =
   None
 | Some of 'a
```

permet de représenter le fait qu'un élément est "présent" ou "existant" (par exemple `Some 42`) ou qu'il n'existe pas (`None`).

3.1 Listes d'association

Une liste d'association est une liste de paires (clé, valeur). Par exemple, la liste

```
let exemple_assoc = [("Max", 10); ("Nicolas", 4); ("Nicole", 9)]
```

établit une association entre le nom et l'age d'une personne.

Exercice 5

1. Écrivez une fonction `lookup_assoc` qui prend une clé et une liste d'association et renvoie `Some` (valeur associée) si la clé apparaît dans la liste, et `None` sinon.

   Exemple:
   `lookup_assoc "Nicolas" [("Max", 10); ("Nicolas", 4); ("Nicole", 9)]` donne `Some 4`,
   `lookup_assoc "Maurice" [("Max", 10); ("Nicolas", 4); ("Nicole", 9)]` donne `None`

2. Écrivez une fonction `add_assoc` qui prend une clé, une valeur et une liste d'association, et ajoute la paire (clé, valeur) à la liste.

3. Écrivez une fonction `remove_assoc` qui prend une clé et une liste d'association et supprime la (les) valeur(s) associée(s).

Il existe plusieurs manières d'implanter ces fonctions. Discutez les avantages et inconvénients. Vos fonctions doivent avoir le type le plus général possible, c.à.d., pas être uniquement applicables aux listes d'associations `string * int`.

Toutes les implantations doivent satisfaire certains invariants, par exemple: `lookup_assoc k (add_assoc k v kvs) = Some v`. Trouvez d'autres invariants de vos fonctions! Testez vos fonctions pour vérifier qu'ils sont satisfaits.

3.2 Maps

Un map est une fonction qui associe une valeur (option) à une clé:

```
type ('k, 'v) map = 'k -> 'v option
```

Les maps peuvent être considérés comme des implantations spécifiques d'associations.

Exercice 6

1. Écrivez le map `exemple_map` qui correspond à la liste d'association `exemple_assoc` plus haut.

2. Écrivez les fonctions `lookup_map, add_map, remove_map` qui correspondent aux fonctions de l'exercice précédent.

3. Écrivez une fonction `empty_map`, qui renvoie `None` pour tout argument.

4. Écrivez une fonction `assoc2map` qui convertit une liste d'association en map et qui s'appuie sur `empty_map` et `add_map`. Testez vos fonctions, par exemple:

   ```
   # lookup_map "Nicole"
     (assoc2map [("Max", 10); ("Nicolas", 4); ("Nicole", 9)]) ;;
   - : int option = Some 9
   ```

4 Règles de typage élémentaires

Exercice 7 (Typage) Écrivez la fonction `tp_of_expr` qui prend deux arguments: un contexte et une expression, et qui renvoie le type de cette expression. Puisque l'expression peut être mal typée, votre fonction doit gérer ce cas et répondre avec un message approprié. Il suffit de lever une exception qui explique les raisons de l'échec du typage. Inspirez-vous des messages produits par Caml dans une situation similaire.

Exemple: l'expression correspond à `(fun (x: int) -> x) 2`:

```
# tp_of_expr [] (App (Abs ("x", ConstT IntT, Var "x"), Const (IntE 2))) ;;
- : tp = ConstT IntT
```

Exemple: l'expression correspond à `(fun (x: int) -> x) true`:

```
# tp_of_expr [] (App (Abs ("x", ConstT IntT, Var "x"), Const (BoolE true))) ;;

Exception:
Failure
"(fun (x:int) -> x) !! true !!
 This expression has type bool but an expression was expected of type int".
```

Ici, l'expression incriminée se trouve entre `!!  ...  !!`. Caml souligne l'expression.

5 Extension de la syntaxe et des règles du typage

Exercice 8 (Paires) On souhaite représenter le type des paires, de la forme `T * T'`, et des paires, de la forme `(e, e')`. Modifiez les types `tp` et `expr` pour permettre la représentation de termes comme `fun (x: int) -> fun (y: int) -> (x, y)` dont la vérification de type devrait donner la représentation de `int -> int -> int * int`.

Adaptez les fonctions d'affichage `string_of_type` et `string_of_expr`.

Adaptez la fonction `tp_of_expr`.

Exercice 9 (Opérateurs) On souhaite représenter des opérateurs arithmétiques ou booléens, par exemple écrire `fun (x: int) -> fun (y: int) -> (x + y) > y`. On va se limiter à des opérateurs binaires.

Modifiez le type `expr` pour prendre en compte des opérateurs. Veillez à une définition modulaire qui

- permet de rajouter facilement d'autres opérateurs (conséquence: n'introduisez pas un constructeur de `expr` par opérateur, mais factorisez les opérateurs dans un nouveau type)
- classe les opérateurs par leur type, afin de faciliter l'écriture des règles de typage *Exemple:* `+` prend deux `int` et produit un `int`, et pareil pour `-`, tandis que `>` prend deux `int` et produit un `bool`.

Exercice 10 (Match) Concevez un type pour des expressions `match`, de la forme

```
match e with
  C1(args1) -> e1
| ...
| Cn(argsn) -> en
```

où les `Ci` sont des constructeurs, `argsi` sont des listes de variables (on se passe d'expressions de filtrage plus complexes) et `ei` des expressions.