Feuille d'exercices n°1

Exercice I : Typage et évaluation d'expressions

Q1 – Pour chacune des expressions suivantes, dire si elle est une expression valide du langage OCaml, dans le cas où l'expression est valide, donner son type et sa valeur ainsi que les étapes de calcul de cette valeur dans le modèle équationnel.

1) `(3 * 5) + 2`

2) `((fun x -> 2 * x) 7)`

3) `((fun x -> 2 * x) (3 * 5))`

4) `(((fun x -> (fun y -> x * y)) 3) 7)`

5) `((fun x -> fun y -> x * y) 3 7)`

6) `(fun f -> fun x -> (f x) + x) (fun x -> x - 1) 3`

7)
```
let (x, y) = (1, 2) in
x + y
```

8)
```
let a = (1, 2) in
let (x, y) = a in
x + y
```

9)
```
let f x = x + 1 in
(f, 3)
```

10)
```
let b = true in
if b then (fun x -> x)
else 0
```

11)
```
let b = true in
((if b then (fun x -> x + 1)
else (fun x -> x - 1)) 5)
```

12)
```
((fun b ->
  ((fun x -> fun y -> 2 * x + 3 * y) (if b then 1 else 2) 5)
  ) true)
```

13)
```
let f = fun x -> not x in
let g = fun f -> fun x -> fun y -> (f (not x)) && (f y) in
(g f true false)
```

Exercice II : N-uplets

Q1 – Définir les fonctions `fst: ('a * 'b) -> 'a` (resp. `snd: ('a * 'b) -> 'b`) projetant une paire sur sa première (resp. seconde) composante.

Q2 – Définir une fonction `paire: 'a -> 'b -> ('a * 'b)` qui, à partir de deux éléments a et b construit la paire (a, b).

Q3 – Déduire de la question précédente une fonction `paire_true: 'a -> (bool * 'a)` qui a un élément a associe la paire (`true`, a).

Q4 – Définir une fonction `curry: (('a * 'b) -> 'c) -> ('a -> 'b -> 'c)` prenant en argument une fonction `f` attendant une paire comme argument, et calculant une fonction de deux arguments `g` telle que pour tout `x`, `y`, on ait `(g x y) = (f (x, y))`.

Q5 – Définir une fonction `uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c)` prenant en argument une fonction `g` à deux arguments, et calculant une fonction `f` attendant une paire telle que pour tout `x`, `y`, on ait `(f (x, y)) = (g x y)`.

Exercice III : Manipulation de fonctions

En programmation fonctionnelle, les fonctions sont des valeurs comme les autres et en particulier, il est possible d'écrire des fonctions prenant des fonctions en paramètres et/ou produisant des fonctions en sorties.

Q1 – Écrire la fonction `application` qui prend en argument une fonction `f` et une valeur `x` et qui applique `f` à `x`.

Q2 – Écrire la fonction `composition` prenant en argument deux fonctions `f` et `g` et calculant leur composée `f` ∘ `g`.

Q3 – Écrire la fonction `f_ou_ident` prenant en argument une fonction `f` et un booléen `b` et calculant

- l'identité si `b` est vrai
- `f` si `b` est faux

Travaux sur machines

Configuration

Nous utiliserons l'interpréteur de code-octet `utop` pour évaluer le code OCaml demandé en TP. Cet interpréteur interactif permet de saisir directement les expressions à évaluer, de charger un fichier source contenant des définitions de fonctions, d'évaluer l'application des fonctions chargées, etc.

Les fichiers sources peuvent être réalisés avec n'importe quel éditeur de texte. On peut conseiller, parmi ceux-ci, l'éditeur `atom` qui fournit un bon support pour le langage OCaml (syntaxe, typage, etc.). Vous pouvez toutefois utiliser un autre éditeur de texte mais il est alors très fortement recommandé d'en choisir un qui supporte OCaml (voir `https://github.com/ocaml/merlin/wiki`).

Nous donnons ci-dessous quelques consignes pour configurer votre compte pour développer en OCaml dans le cadre de cette UE.

Veuillez taper dans un terminal les deux commandes suivantes :

```
$ sh /Infos/lmd/2020/licence/ue/LU2IN019-2020oct/install.sh
$ source ~/.bashrc
```

Si vous choisissez d'utiliser `atom`, pour installer les plugins `OCaml`, tapez dans un terminal la commande suivante

```
$ sh /Infos/lmd/2020/licence/ue/LU2IN019-2020oct/atom-setup.sh
```

Interpréteur `utop`

Pour ouvrir un interpréteur (*toplevel*) OCaml dans le terminal :

```
$ utop
```

Vous devez obtenir quelque chose qui ressemble à

```
-( 15:02:02 )-< command 0 >-----------------------------------{ counter: 0 }-
utop #
```

Après l'invite (*prompt*) `utop #`, vous pouvez entrer les expressions à évaluer. Par exemple:

```
-( 15:02:02 )-< command 0 >-----------------------------------{ counter: 0 }-
utop # (14 * 3);;
```

Notez le double point-virgule (`;;`) après l'expression qui lance l'évaluation.

La réponse de l'interpréteur doit être:

```
- : int = 42
-( 15:02:02 )-< command 1 >-----------------------------------{ counter: 0 }-
utop #
```

qui indique le type et la valeur de l'expression. L'interpréteur est prêt pour une autre évaluation.

Pour charger un fichier source, utilisez la commande `#use` (avec un `#` en tête)

```
utop # #use "toto.ml" ;;
```

Le contenu du fichier source `toto.ml` est compilé. Tous les types et toutes les fonctions qui y sont définies deviennent utilisables.

Par exemple, si le fichier `toto.ml` contient la définition

```
let f (x: int) (y:int) : int =
   x + y + 2
```

après sont chargement, vous pourrez appliquer la fonction `f`:

```
utop # #use "toto.ml" ;;
-( 15:02:02 )-< command 0 >-----------------------------------{ counter: 0 }-
utop # (f 2 3);;
- : int = 7
```

Cette interaction avec `utop` vous servira à mettre au point les réponses aux questions posées.

Vous devez donner des jeux de tests pour vos fonctions. Pour cela, vous pourrez utiliser la primitive `assert` connue en OCaml. Par exemple:

```
-( 15:02:02 )-< command 1 >-----------------------------------{ counter: 0 }-
utop # assert ((f 2 3) = 7);;
- : unit = ()
```

Notez les parenthèses obligatoires autour de `((f 2 3) = 7)`. Sans elles, vous obtenez un message d'erreur.

Si le test est correct la valeur obtenue est la valeur spéciale notée `()` de type `unit`. Lorsque le test ne passe pas, l'interpréteur répond en déclanchant une exception: `Assert_failure`. Par exemple:

```
-( 15:02:02 )-< command 2 >-----------------------------------{ counter: 0 }-
utop # assert ((f 2 3) = 42);;
Exception: Assert_failure ("//toplevel//", 1, 0).
```

Ce qui suit lee nom de l'exception indique où l'erreur s'est produite.

Rendu de TP

Vous soumettrez les réponses aux exercices de TP sous forme d'un ficheir source `.ml` contenant les définitions de fonctions et de types demandés. Vous commenterez votre code source en indiquant vos idntifiant, de quelle séance de TP il s'agit, de quelle question il s'agit, etc.

Exemple:

```
(* ======================================== *)
(* == LU2IN019 - P. Manoury              == *)
(* == TP 0                               == *)
(* ======================================== *)

(* == Question 1 *)
let f (x: int) (y:int) : int =
  x + y + 2;;

let _ = assert ((f 2 3) = 7)
let _ = assert ((f 0 0) = 2)
let _ = assert ((f (-1) (-1)) = 0)
```

Soumettez un seul fichier par séance de TP contenant toutes les réponses que vous pouvez donner.

Exercice IV : Addition binaire

Dans cet exercice nous allons définir un additionneur pour les nombres entiers binaires. Nous nous limiterons aux nombres codables sur *4 bits* qui corespondent aux entiers compris entre 0 et 15. Les *bits* sont représentés par les valeurs booléennes `true` et `false`. Par exemple, l'entier 13 (1101 en binaire)[1] est codé par le quadruplet `(true,true,false,true)`.

Afin de faciliter la manipulation de tels entiers nous allons définir de nouveaux types que nous pourrons ensuite utiliser dans la description de nos fonctions. Une définition de type se fait en OCaml à l'aide du mot clé `type`. Par exemple, le type `duet` (pour 2 *bits*) est défini par : `type duet = bool * bool`.

Q1 – Définir le type `bit` comme étant un booléen, le type `duet` comme étant une paire de booléen et finalement le type `quartet` comme étant un quadruplet de booléen.

Nous allons commencer par définir un *demi-additionneur*. Un demi-additionneur prend en argument deux bits a et b et calcule deux bits s et r de sorte que s est le résultat de l'addition de a et de b modulo 2 et r est la retenue résultant de l'addition de a et b.

Le tableau ci-dessous donne une définition par extension du demi-additionneur :

a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Avec la vision booléenne des bits (`true` pour 1 et `false` pour 0) on peut remarquer que s est le ou-exclusif (xor) de a et b.

[1] Rappel: $13 = 8 + 4 + 1 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

Q2 – Définir la fonction `xor` de signature `bit -> bit -> bit` et calculant le ou-exclusif de ses 2 arguments.

Q3 – Définir une fonction `half_adder` de signature `bit -> bit -> (bit * bit)` qui étant donné a et b calcule la paire (s, r) comme indiqué dans le tableau ci-dessus.

Étant donné que l'addition de deux bits peut produire une retenue il est claire que nous allons avoir besoin d'un additionneur ne calculant pas seulement la somme et la retenue de deux bits mais plutôt d'un additionneur calculant la somme et la retenue de trois bits (deux bits provenant des nombres dont nous souhaitons calculer la somme et un étant la retenue de l'addition précédante).

Ainsi étant donné trois bits a b et c un additionneur de ces trois calculera la somme de ces trois bits et une retenue en :

- demi-additionnant a et b, produisant une somme s_1 et une retenue r_1;
- demi-additionnant c et s_1, produisant une somme s_2 et une retenue r_2;

la somme est alors s_2 et la retenue sera non nul si et seulement si une des retenues (r_1, r_2) est non nul.

Q4 – Définir une fonction `adder` de signature `bit -> bit -> bit -> (bit * bit)` et calculant l'addition de ses 3 arguments.

Q5 – Tester (de manière exhaustive) votre fonction `adder`.

Q6 – En déduire une fonction `duet_adder` de signature `duet -> duet -> bit -> (duet * bit)` calculant la somme et la retenue des deux duets et de la retenue qui lui sont passés en argument.

Q7 – En déduire une fonction `quartet_adder` de signature `quartet -> quartet -> bit -> (quartet * bit)` calculant la somme et la retenue des deux quartets et de la retenue qui lui sont passés en argument.

Q8 – Définir une fonction `to_quartet` de signature `int -> quartet` prenant en argument un entier entre 0 et 15 et calculant sa représentation comme un quadruplet de booléen.

Q9 – Utiliser la fonction `to_quartet` pour tester votre fonction `quartet_adder`.