Examen de
λ-calcul, typage et algorithmique
Durée : 2 h
Tous documents autorisés

Ce sujet comporte 4 pages.

1 λ-calcul typé

1. Soient les λ-termes suivants :

```
#let xyz = fun x y z c -> c x y z;;

#let f = fun c -> c (fun x y z -> x);;

#let s = fun c -> c (fun x y z -> y);;

#let t = fun c -> c (fun x y z -> z);;

#let x = xyz "un" (xyz (xyz 1 2.0 "trois") 2 3.0) '3' in t (f (s x));;
```

   (a) Typer `xyz`, `f`, `s` et `t`.

   (b) Évaluer la dernière expression.

2. Pour construire des tuples de taille arbitraire, on enrichit le λ-calcul avec une construction « n-uplet » $(a_1, \ldots, a_n)$ et n constructions « projection » $\texttt{proj}_n^i$, pour tout entier $n \geq 0$. La règle de réduction est donc :

$$\texttt{proj}_n^i\,(a_1, \ldots, a_n) \to a_i$$

   Le type d'un n-uplet sera noté[1] $\tau_1 \times \ldots \times \tau_n$. Il suffit alors d'ajouter les $(n+1)$ règles de typage suivantes pour pouvoir typer des expressions où figurent des n-uplets :

$$\frac{\Gamma \vdash a_1 : \tau_1 \;\ldots\; \Gamma \vdash a_n : \tau_n}{\Gamma \vdash (a_1, \ldots, a_n) : \tau_1 \times \ldots \times \tau_n} \times_n \qquad \frac{\Gamma \vdash a : \tau_1 \times \ldots \times \tau_n}{\Gamma \vdash \texttt{proj}_n^i\, a : \tau_i}\texttt{proj}_n^i$$

   pour $1 \leq i \leq n$.

   (a) Que peut-on dire du cas où $n = 0$? Donner une valeur de ce type et les règles de typage associées.

   (b) Typer (en détaillant l'application des règles) et évaluer l'expression suivante :

```
proj3^2 ((fun () -> 1), (fun () -> "deux"), (fun () -> 3.0)) ()
```

2 Algorithmique

On veut étudier les flux d'avions entre secteurs de contrôle. On modélise un centre de contrôle par un graphe non-orienté dont les nœuds sont les secteurs élémentaires, étiquetés par leur nom, et dont les arêtes sont des flux d'avions entre deux secteurs, étiquetées par le nombre d'avions (qui passent d'un secteur à l'autre).

[1] Il faudrait noter $\times_n$ au lieu de $\times$ pour distinguer rigoureusement les constructeurs de type d'arités différentes.

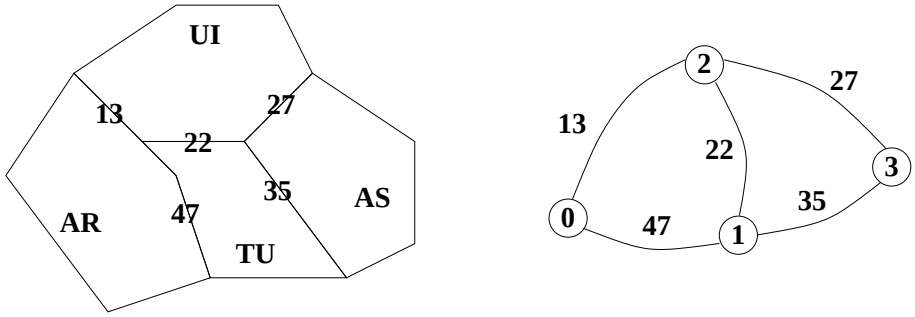

FIG. 1 – Un centre de quatre secteurs et le graphe des flux associé. `AR` correspond au nœud 0, `TU` à 1, `UI` à 2 et `AS` à 3.

On dispose de données sous la forme d'une liste de triplets

```
#type data = (string * string * int) list;;
```

correspondant à l'ensemble des arêtes du graphe. Les deux premiers éléments sont les noms des secteurs et le troisième la taille du flux entre ces deux secteurs.

Pour manipuler plus facilement le graphe, on va le transformer en une liste de listes d'adjacences et remplacer les chaînes de caractères utilisées pour identifier les secteurs par des entiers (de 0 à $n-1$ si n est le nombre de secteurs, cf. figure 1).

On utilise d'abord une représentation intermédiaire : une table de hachage dont les clés seront des chaînes de caractères (les noms des secteurs) et les données seront du type :

```
#type node = {id : int; adj : (int * int) list ref};;
```

Le champ `id` correspond au nouveau nom de type `int` du secteur et `adj` à une référence vers sa liste d'adjacences en cours de construction. La liste d'adjacences est une liste de couples `(secteur_id, nb_avions)`.

On va alors remplir la table en parcourant la liste des données de la manière suivante. Pour chaque nouvelle arête, on vérifie si les secteurs existent déjà dans la table. S'ils existent, on met à jour les références sur leur liste d'adjacences en ajoutant en tête de liste le couple représentant la nouvelle arête. Si on est en présence d'un secteur encore inconnu, il faut d'abord lui attribuer un nouvel identificateur entier et l'ajouter à la table.

1. Soit la fonction suivante :

```
#let gen_int = let n = ref (-1) in fun () -> incr n; !n;;
```

   Quelle est son type ? Quelles sont les valeurs renvoyées par les trois premiers appels à `gen_int` avec l'argument `()` ?

2. Soit `ht` le type des tables de hachage décrites précédemment, écrire la fonction
   `find_or_create_node : ht -> string -> node`
   qui prend en arguments une table de hachage et le nom d'un secteur.
   `find_or_create_node table name` doit renvoyer la donnée associée à `name` dans `table` si elle existe. Sinon, elle crée un nouveau nœud de type `node` avec un nouvelle identificateur entier et une référence vers une liste d'adjacences vide, puis elle ajoute cette entrée dans `table` et renvoie ce nouveau nœud. On utilisera la fonction `gen_int` précédente ainsi que la librairie standard d'OCaml (pour manipuler les tables de hachage).

3. Écrire la fonction
   `add_edge : ht -> (string * string * int) -> unit`
   `add_edge table edge`[2] met à jour les listes d'adjacences des extrêmités de l'arête `edge` dans `table` en *modifiant en place* leur référence. On utilisera la fonction `find_or_create_node` précédente.

[2] On rappelle que « edge » signifie « arête » en Anglais et « node », « nœud ».

4. Écrire la fonction
   `edges2adj : data -> (int * (int * int) list) list`
   qui transforme dans un premier temps les données en une table de hachage remplie à l'aide de la fonction `add_edge`, puis qui transforme cette table en une liste de listes d'adjacences qui sera finalement renvoyée. Pour cette dernière transformation, on pourra utiliser la fonction `Hashtbl.iter` ou, mieux, la fonction[3]

```
#Hashtbl.fold;;
- : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) Hashtbl.t -> 'c -> 'c = <fun>
```

   qui est l'équivalent pour les tables de hachage de la fonction `List.fold_right` pour les listes.

```
#let data = [("AR", "TU", 47); ("UI", "TU", 22); ("AR", "UI", 13);
#            ("TU", "AS", 35); ("AS", "UI", 27)];;

#edges2adj data;;
- : (int * (int * int) list) list =
[(1, [(3, 35); (2, 22); (0, 47)]); (3, [(2, 27); (1, 35)]);
 (0, [(2, 13); (1, 47)]); (2, [(3, 27); (0, 13); (1, 22)])]
```

5. Estimer la complexité temporelle de `edges2adj data` en fonction du nombre d'arêtes du graphe. On considérera que l'ajout ou la consultation d'une entrée dans la table de hachage prend un temps constant. Pourrait-on faire diminuer cette complexité avec une technique plus performante ?
6. On veut calculer la somme de tous les flux traversant chaque secteur. Écrire une fonction qui transforme la liste de listes d'adjacences en une liste de couples `(secteur_id, nb_total_avions)`.
   Estimer la complexité de ce calcul en fonction du nombre de nœuds et du degré δ du graphe (le degré d'un nœud est le nombre de ses voisins et celui d'un graphe est le degré maximal de ses nœuds).

Appendice : quelques fonctions du module Hashtbl

```
type ('a, 'b) t
The type of hash tables from type 'a to type 'b.

val create : int -> ('a, 'b) t
Hashtbl.create n creates a new, empty hash table, with initial size n.
For best results, n should be on the order of the expected number of
elements that will be in the table. The table grows as needed, so n is
just an initial guess.

val add : ('a, 'b) t -> 'a -> 'b -> unit
Hashtbl.add tbl x y adds a binding of x to y in table tbl. Previous
bindings for x are not removed, but simply hidden. That is, after
performing Hashtbl.remove tbl x, the previous binding for x, if any,
is restored. (Same behavior as with association lists.)

val find : ('a, 'b) t -> 'a -> 'b
Hashtbl.find tbl x returns the current binding of x in tbl, or raises
Not_found if no such binding exists.

val mem : ('a, 'b) t -> 'a -> bool
Hashtbl.mem tbl x checks if x is bound in tbl.
```

[3]Disponible à partir de la version 3.02 du compilateur OCaml.

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
Hashtbl.iter f tbl applies f to all bindings in table tbl. f receives
the key as first argument, and the associated value as second
argument. The order in which the bindings are passed to f is
unspecified. Each binding is presented exactly once to f.

val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
Hashtbl.fold f tbl init computes (f kN dN ... (f k1 d1 init)...),
where k1 ... kN are the keys of all bindings in tbl, and d1 ... dN are
the associated values. The order in which the bindings are passed to f
is unspecified. Each binding is presented exactly once to f.
```