TD5 : Typage avancé

Dans cette section, on s'intéresse à l'expression de propriétés des valeurs d'un langage (ici OCaml) dans le système de types, afin que le compilateur/interprète puisse décharger la preuve à notre place et puisse également mettre en place des optimisations liées à la connaissance de ces propriétés. Tout ce qui suit est implantable, au moins en partie, dans les langages *mainstream* qui supportent la généricité paramétrique, par exemple C++, Java, Rust, ADA, etc.

1 Types fantômes

Les types fantômes sont des types paramétrés dont un paramètre n'apparaît pas dans la définition du type. Ce paramètre est donc inutile au sens où il ne représente aucune donnée associée au type défini. Par contre, il peut représenter d'autres informations utiles, comme l'état "interne" d'une valeur de ce type. Le paramètre est donc vu comme une annotation gérée par le compilateur/interprète, à la différence d'un commentaire. On peut notamment décrire des machines à états, des protocoles, etc.

Dans l'exemple suivant, on décrit le type des fichiers dans lesquels on doit lire un unique caractère, au moyen d'un automate à 2 états `debut` et `fin`.

```
module type FichierLecture1Car =
sig
  type debut
  type fin
  type _ fichier
  val open_ : string -> debut fichier
  val read : debut fichier -> char * fin fichier
  val close : fin fichier -> unit
end
```

Il est important que les utilisateurs aient accès aux type `_ fichier` en tant que type abstrait à travers une interface de module uniquement et non directement à l'aide de constructeurs du type. Dans ce dernier cas, en définissant par exemple `type _ fichier = in_channel`, le paramètre du type ne jouant aucun rôle, on pourrait tout à fait écrire une fonction de transtypage `magic: 'a fichier -> 'b fichier` qui réduirait à néant la contrainte d'accès au fichier (ouverture → lecture → fermeture) qui garantit qu'on ne peut pas lire un fichier fermé, etc.

Une implantation possible est la suivante, dans laquelle on constate que les types fantômes ne jouent aucun rôle :

```
module Impl : FichierLecture1Car =
struct
  type debut = unit
  type fin = unit
  type _ fichier = in_channel
  let open_ nom = open_in nom
  let read f = (input_char f, f)
  let close f = close_in f
end
```

Pour que ceci soit utile, il est nécessaire d'imposer un usage purement séquentiel d'un fichier, sinon on peut lire un nombre quelconque de caractères en parallélisant, i.e. en écrivant par exemple :

```
let wrong = let f = Impl.open "toto" in (Impl.read f, Impl.read f, ...)
```

Le système de type ne se plaindra pas dans ce cas, ce qui constitue une faiblesse de l'approche. Néanmoins, en pratique, la plupart des codes sont naturellement séquentiels.

On peut quand même garantir cette propriété (dite de linéarité) en encapsulant l'accès au fichier par une approche monadique. Voici un exemple de code conforme "linéaire", dans lequel les deux variables successives `f` ont pour type `debut fichier` puis `fin fichier` :

```
let lire_char nom =
  let f = Impl.open_ nom in
  let (c, f) = Impl.read f in
  (Impl.close f; c);;
```

▷ **Exercice 1** *On veut étendre/adapter l'interface précédente au cas où on veut lire deux caractères, puis au cas où on veut lire un nombre pair de caractères.*

2 Types non uniformes

Une définition récursive de type `'a t` peut faire apparaître des instances différentes du paramètre `'a`, ce qui correspondra naturellement à des fonctions récursives définies et appliquées sur ces instances différentes. De tels types et fonctions sont appelés **non-uniformes**. Considérons par exemple le type suivant `'a perfect_tree` dans lequel on remarque que le constructeur `Node` possède un argument d'une instance différente `('a * 'a) perfect_tree`. Ce type représente les "listes" contenant successivement un élément, une paire d'éléments, une paire de paires, etc. Cette description correspond également aux arbres binaires parfaitement équilibrés, si l'on considère le nombre d'éléments rangés par profondeur croissante.

```
type 'a perfect_tree = | Empty | Node of 'a * ('a * 'a) perfect_tree;;
```

Cet invariant structurel permet de définir des opérations pour fusionner ou couper en deux de tels arbres parfaits. Elles feront appel à la récursion non-uniforme. Pour cela, il faut annoter le type de la fonction définie et abstraire dans ce type les paramètres d'instances qui changent d'un appel à l'autre, au moyen de la notation : `type parametre.` *type_de_la_fonction*.

▷ **Exercice 2** *Définir la fonction* `split : ('a * 'a) perfect_tree -> 'a perfect_tree * 'a perfect_tree`.

▷ **Exercice 3** *Quelle information (de type) manque-t-il à* `'a perfect_tree` *pour pouvoir définir la fonction suivante, réciproque de* `split` *?*

```
merge : 'a perfect_tree -> 'a perfect_tree -> ('a * 'a) perfect_tree
```

Modifier le type `'a perfect_tree` *et la fonction* `split` *en conséquence. Définir enfin la fonction* `merge`.

3 Generalized Algebraic Datatypes(GADT)

Le mécanisme de *Run-Time Type Information* (RTTI) présent dans la plupart des langages orientés objet, à divers dégrés, permet de représenter les types à l'exécution. Citons par exemple le `instanceof` de JAVA ou la liaison tardive qui utilisent le RTTI. On peut facilement émuler ceci à l'aide de GADT, tout en garantissant la sûreté du code. Voici un exemple de représentation de valeurs avec leur type, en l'occurence les constantes entières et l'addition d'entiers. On construit également une fonction d'évaluation triviale :

```
type _ repr =
| Int : int -> int repr
| Add : (int -> int -> int) repr

let eval : type a. a repr -> a =
  function
  | Int i -> i
  | Add   -> (fun a b -> a+b)
```

▷ **Exercice 4 (Généralisons !)**
On souhaite représenter et évaluer des expressions entières ne contenant que l'opération d'addition (par exemple "1+(2+3)").

- *Ajouter la possibilité a minima d'appliquer l'addition sur deux arguments entiers.*
- *Généraliser en permettant d'appliquer toute fonction sur tout argument (même si on ne dispose pour l'instant que de l'addition).*
- *Généraliser encore en permettant de représenter des valeurs de tout type.*
- *Généraliser enfin en permettant l'ajout de variables (question bonus).*

Grâce aux GADT, on peut également envisager la situation inverse des types fantômes, i.e. quand une variable de type apparaît dans les arguments d'un constructeur mais pas dans le type lui-même. Ceci correspond à un type existentiel, qui permet la génération de types uniques. Ainsi, le type ci-dessous représente-t-il une façon universelle d'encapsuler toute valeur de tout type. Néanmoins, lorsqu'on filtre une valeur du type `univ` pour récupérer la valeur fournie à la création, on ignore totalement son type, qui sera vu par le système de types, de façon sûre, comme une nouvelle constante de type unique, ce qui rend la valeur extraite complètement inutilisable ! Il faut donc avoir un vrai type abstrait, comme `stringable`, avec des primitives utiles, pour pouvoir exploiter ce mécanisme de type existentiel. On vérifie qu'on a bien défini un type existentiel car on a `Any : ∀ a. a→univ ≡ (∃ a. a)→univ`

```
(* encapsulation universelle inutile *)
type univ = Any : 'a -> univ
(* encapsulation universelle des types pouvant etre convertis en chaine *)
type stringable = Stringable : 'a * ('a -> string) -> stringable

let string_of_stringable (Stringable (v, f)) = f v
let string_of_char c = String.make 1 c
let string_of_list f l = "[" ^ String.concat "; " (List.map f l) ^ "]"
(* liste heterogene, contenant des int, char et int list *)
let hliste = [Stringable (42, string_of_int);
              Stringable ('a', string_of_char);
              Stringable ([1;2;3], string_of_list string_of_int)]
let string_of_hliste hl = string_of_list string_of_stringable hl

string_of_hliste hliste
```

Remarque : Les types existentiels existent déjà au niveau des modules, ils correspondent simplement au mécanisme des types abstraits. Ainsi, l'implémentation par un module `Mod` d'une signature de type abstrait `Sig` contenant une déclaration `type t` équivaut à la création d'une constante de type `Mod.t` unique, dont la définition est cachée.