Programmation avancée

TD6 : Typage avancé : GADTs

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

Adapté du sujet de Matthieu Hilaire — transcrit par Xavier Denis

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `failwith "TODO"`.

Pensez également à tester votre code avec `dune test`, et en décommentant les tests au fur et à mesure.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/06-gadts/td06.zip`

Rappel : lorsque l'on écrit des fonctions polymorphes manipulant des GADTs, il faut en général utiliser la syntaxe inhabituelle suivante :

```
let mafonction : type a b c. ... -> ... = fun x y z ->
  ...
```

1 Échauffement

fichier `lib/partie1.ml`

Une fonctionnalité essentielle des GADTs est la possibilité pour un constructeur de *quantifier existentiellement* sur des variables de type.

Intuitivement, il est possible de définir des types GADTs correspondant à des types de la forme « `exists type a. ...` » — où la construction `exists type` (qui n'existe pas en OCaml) correspondrait à une quantification existentielle.

Considérons par exemple le type GADT suivant :

```
type some_list = SomeList : 'a list -> some_list
```

Dans cette déclaration, la variable de type `'a` est quantifiée existentiellement : le type `some_list` correspond moralement au type « `exists type a. a list` », c'est à dire au type des listes de `a` pour un certain type `a` inconnu.

Exercice 1. *Implémenter une fonction* `some_rev : some_list -> some_list` *qui renverse l'ordre des éléments de la liste passée en argument. Noter que l'on peut implémenter cette fonction alors que l'on ne connaît rien sur le type des éléments de la liste !*

Considérons maintenant le GADT :

```
type dyn = Dyn : 'a -> dyn
```

Exercice 2.

1. *Intuitivement, quel est l'ensemble des valeurs OCaml représentées par le type* `dyn` *?*
2. *Peut-on utiliser* `dyn` *pour créer une liste contenant des valeurs de types différents (entiers, booléens, chaînes de caractères), en utilisant le type standard des listes ? (si oui, le faire)*
3. *Étant donné une valeur de type* `dyn`*, que peut-on faire avec ?*

On veut maintenant définir un GADT plus intéressant, que l'on appellera `printable`. Une valeur du type `printable` correspond à un type OCaml `'a` (quantifié existentiellement), une valeur du type `'a`, et une fonction d'affichage pour cette valeur (de type `'a -> string`).
Intuitivement, `printable` correspond au type « `exists type a. a * (a -> string)` ».

```
type printable = ...
```

Exercice 3.

1. *Définir* `printable`
2. *Implémenter une fonction d'affichage pour les éléments de* `printable`*, de type:* `printable -> string`
3. *Implémenter des fonctions permettant de convertir des entiers et chaines de caractères en valeurs* `printable` *: c'est à dire des fonctions de type* `int -> printable` *et* `string -> printable`*.*

Une autre fonctionnalité essentielle des GADTs est la possibilité pour un constructeur d'indiquer des *égalités entre types*.

Notamment, on peut définir un GADT correspondant exactement à un témoin d'égalité entre deux types :

```
type (_, _) eq =
  | Eq : ('a, 'a) eq
```

Si l'on a une valeur de type `(a, b) eq`, alors en utilisant le pattern-matching sur cette valeur, on apprend que l'on est forcément dans le cas `Eq` et que les types `a` et `b` sont en fait égaux.

Exercice 4. *Implémenter la fonction* `convert` *de type* `('a, 'b) eq -> 'a -> 'b`*, qui permet de convertir d'un type à un autre étant donné un témoin d'égalité entre ces types.*

Exercice 5. *En utilisant* `convert`*, écrire une fonction* `convert_list` *de type* `('a, 'b) eq -> 'a list -> 'b list`.

Si ce n'est pas déjà le cas, faire en sorte que `convert_list` *s'exécute en temps constant !*

Exercice 6. *Étant donné une valeur de type* `('a list, 'b list) eq`*, peut-on obtenir une valeur de type* `('a, 'b) eq` *? Si oui, implémenter une telle fonction. Est-ce possible pour un type abstrait inconnu à la place de* `list` *(et pourquoi ?) ?*

On considère les deux GADTs suivants :

```
type _ t =
  | Int : int -> int t
  | Bool : bool -> bool t

type _ ty =
  | IntTy : int ty
  | BoolTy : bool ty
```

Exercice 7. *Quelle est la différence entre une valeur de type* `'a t` *et une valeur de type* `'a ty` *? Pour y voir plus clair, on pourra construire des valeurs de type* `'a t` *et* `'a ty`*.*

Donner une version non-GADT de ces deux type, qu'on appellera `t'` *et* `ty'`*.*

On dit en général que `'a ty` est une « représentation de type à l'exécution », car une valeur `'a ty` nous permet de savoir à l'exécution ce qu'est le type `'a` (par pattern-matching), mais ne contient pas de données à proprement parler.

Exercice 8. *Implémenter les fonctions suivantes :*

1. `let default : 'a ty -> 'a`*, qui fournit une valeur par défaut pour le type spécifié par son argument.*
2. `let gettype : 'a t -> 'a ty`*, qui renvoie la description du type (entier ou booléen) de son argument.*
3. `let iseqty : 'a ty -> 'b ty -> ('a, 'b) eq option`*, qui teste si deux représentations de types sont égales, et renvoie dans ce cas un témoin d'égalité des types OCaml.*

2 Interprète et typeur fortement typés

fichier `lib/partie2.ml`

On considère un mini-langage suivant, ainsi que le code permettant de l'interpréter (inclus dans le code fourni) :

```
(** AST non typé *)
type expr = ...

(** Valeurs *)
```

```
type value = ...

(** Évaluateur non typé *)
let rec eval (e : expr) : value = ...
```

Exercice 9. *Définir un GADT* `'a wtexpr` *représentant les expressions bien typées, où le paramètre* `'a` *prendra ses valeurs dans* `int`, `string`, *et les produits formés à partir de ceux-ci (par exemple,* `int * string * string`*).*

```
type _ wtexpr = ...
```

`wtexpr` *contiendra les mêmes noms de constructeurs que* `expr`*.*

Exercice 10. *Adapter l'interprète précédent pour en obtenir une version où le typage garantit qu'on obtient une valeur du bon type :*

```
let rec wteval : type t. t wtexpr -> t = ...
```

On définit maintenant un type « singleton » représentant les types de notre mini-langage : le type OCaml `t typ` ne contient qu'une valeur, représentant le type de notre mini-langage correspondant au type OCaml `t`.

```
type _ typ =
  | TInt : int typ
  | TString : string typ
  | TPair : 'a typ * 'b typ -> ('a * 'b) typ
```

On se donne aussi un type « existentiel » : un `iswtexpr` est un `t wtexpr` pour un certain `t`, auquel on attache une représentation de ce type `t`, qui sera utile si on veut calculer dessus :

```
type iswtexpr = IsWT : 't typ * 't wtexpr -> iswtexpr
```

Exercice 11. *Implémenter une fonction de vérification de type qui prend une expression non typée* `expr`*, lève une exception si elle n'est pas typable, et sinon renvoie un* `iswtexpr`*:*

```
let rec typecheck : expr -> iswtexpr = function ...
```

3 Listes annotées par leur taille

fichier `lib/partie3.ml`

On définit comme un GADT le type des listes annotées par leur taille (encodée comme des entiers unaires « fortement typés ») :

```
type z = Z
type 'a s = S of 'a

type ('a, 'n) nlist =
  | [] : ('a, z) nlist
  | ( :: ) : 'a * ('a, 'n) nlist -> ('a, 'n s) nlist

let xyz : (int, z s s s) nlist = [ 1; 2; 3 ]
```

Exercice 12. *Définir la fonction calculant la longueur d'une liste (comme un entier unaire), de type :*

```
let rec length : type a n. (a, n) nlist -> n = ...
```

On cherche maintenant à implémenter un équivalent de la fonction `append` sur les listes :

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x :: l2' -> x :: append l1 l2'
```

Première difficulté : quel type donner à une fonction append sur les listes annotées par leur taille ?

```
let rec append : type a n m. (a, n) nlist -> (a, m) nlist -> (a, ??) nlist =
  ...
```

Il n'est pas possible de directement définir une « fonction d'addition » au niveau des types qui calculerait `n+m`. Cependant, on peut définir un *prédicat* qui relie deux entiers unaires à leur somme.

```
type (_, _, _) sum =
  | SumZ : (z, 'a, 'a) sum
  | SumS : ('a, 'b, 'c) sum -> ('a s, 'b, 'c s) sum
```

Le type `('a, 'b, 'c) sum` est à lire comme l'assertion $a + b = c$, et une valeur de ce type comme la preuve de cette assertion.

Exercice 13. *Donner une « preuve » de* $3 + 2 = 5$*, autrement dit, donner une valeur de type* `(z s s s, z s s, z s s s s s) sum` *:*

```
let proof_3_plus_2 : (z s s s, z s s, z s s s s s) sum = ...
```

Exercice 14. *Implémenter une fonction* `append` *du type suivant :*

```
let rec append : type a n m k.
  (n, m, k) sum -> (a, n) nlist -> (a, m) nlist ->
  (a, k) nlist
=
  ...
```

Cette fonction *demande* en argument une preuve d'addition afin de pouvoir donner un type à sa valeur de retour. Un client de la fonction doit donc être capable de fournir

une telle preuve, ce qui n'est pas toujours possible : `k` n'est pas forcément déjà connu lors de l'appel à `append`.

À la place, on voudrait plutôt que `append` *produise* la preuve que la longueur de la liste résultante est bien la somme des longueurs des listes passées en entrée.

Intuitivement, on aimerait pouvoir définir `append` avec le type suivant :

```
let rec append : type a n m.
  (a, n) nlist -> (a, m) nlist ->
  exists type k. (n, m, k) sum * (a, k) nlist
```

On va donc utiliser un GADT.

Exercice 15. *Définir un GADT* `('a, 'n, 'm) eappend` *exprimant la propriété « il existe un type* `'k`*, tel que l'on a* `('n, 'm, 'k) sum * ('a, 'k) nlist` *».*

Exercice 16. *Implémenter une fonction* `append'` *de type suivant :*

```
let rec append' : type a n m. (a, n) nlist -> (a, m) nlist -> (a, n, m)
eappend
    =
  ...
```

Exercice 17. *Écrire une fonction* `sum_swap` *qui exprime la commutativité de l'incrémentation, c'est-à-dire, une valeur du type* `let rec sum_swap : type n m k. (n, m s, k) sum -> (n s, m, k) sum`

Exercice 18. *Définir la fonction* `rev_append l1 l2` *qui concatène l'inversion de* `l1` *devant* `l2`*. Cette fonction aura donc la signature :* `let rec rev_append : type a n m. (a, n) nlist -> (a, m) nlist -> (a, n, m) eappend`

On définit un type pour les preuves d'égalités comme le GADT suivant :

```
type ('a, 'b) eq = Eq : ('a, 'a) eq
```

Exercice 19. *En utilisant* `Eq`*, prouver l'identité droite de l'addition. C'est a dire une fonction du type :* `let rec sum_0_r : type n k. (n, z, k) sum -> (n, k) eq`

Astuce : Il est important que la fonction soit récursive.

Exercice 20. *Utiliser les questions précédentes pour définir* `rev`*, de type* `let rev : type a n. (a, n) nlist -> (a, n) nlist` *qui inverse la liste donnée.*