https://laurentbloch.org/BlogLB/Creation-d-un-type-de-donnees-les

Création d'un type de données : les tables

- Zinformatiques - Cours de bioinformatique au CNAM -

Date de mise en ligne : dimanche 30 janvier 2005

Le *type* est un concept de base de la programmation. Pour le dire en peu de mots, un type est un nom donné à un ensemble de valeurs parmi lesquelles une variable pourra prendre sa valeur. Ainsi, si je dis *i est une variable de type entier*, cela signifie que les valeurs de *i* seront des nombres entiers, et que la tentative d'affecter à *i* une valeur d'un autre type, caratère par exemple, devra déclencher une erreur.

Outre les valeurs acceptables pour les objets du type, un type définit les *opérations* acceptables : l'addition est acceptable pour les nombres, pas pour les caractères.

Scheme est un langage à typage dynamique, ce qui signifie qu'il est possible d'affecter à une variable préalablement définie dans un type donné (par exemple un nombre) une valeur d'un autre type (une chaîne de caractères par exemple) ; ceci a des avantages et des inconvénients. Parmi les avantages on peut citer la facilité qu'il y a à écrire des procédures capables de traiter des arguments de types variés (cela s'appelle le polymorphisme). Mais cette souplesse a un prix : il convient de vérifier à l'exécution que la variable à laquelle on veut appliquer une procédure a bien le type voulu !

Le bon usage des types contribue fortement à la qualité de la programmation. C'est pourquoi ont été créés des langages à typage statique, comme *Ada*, *Pascal* ou *Java*, qui évitent ainsi de nombreuses erreurs de programmation, mais au prix d'une certaine rigidité.

Pour bénéficier à la fois en *Scheme* des avantages du typage dynamique et de ceux d'un contrôle des types, il convient que le programmeur définisse ses propres types de données, adaptés au sujet de son programme, et qu'il les vérifie à l'exécution.

Voici un petit exemple, où l'on définit un type de données pour des tableaux de dimension 2. On remarquera que les objets contenus dans le tableau sont de type quelconque, et éventuellement hétérogènes, parce que les vecteurs de *Scheme*, choisis pour la réalisation de notre type, prennent des valeurs hétérogènes. D'autre part, il existe des variations autour du type vecteur en Scheme, notamment pour Bigloo, au sujet desquelles on pourra consulter l'article sur les vecteurs de Bigloo qui décrit certaines extensions propres à l'écriture de programmes très efficaces.

Notre type `table` nous donnera :

– un *constructeur*, pour créer des objets du type :

```
(define (make-table n m)
  (let ((the-table (vector "*TABLE*"
                           (make-vector n #f))))
     (do ((i 0 (+ i 1)))
         ((= i n))
         (vector-set! (vector-ref the-table 1)
                      i
                      (make-vector m #unspecified)))
     the-table))
```

Notre type `table` est implanté au moyen d'un vecteur de deux éléments. Le premier élément a pour valeur la chaîne de caractères `*TABLE*`, c'est *l'étiquette* de notre type ; le second élément de ce vecteur est lui-même un vecteur de

vecteurs, qui constitue le tableau proprement dit.

– un *prédicat d'appartenance*, pour vérifier qu'un objet appartient bien au type :

```
(define (table? obj)
  (and (vector? obj)
       (string=? (vector-ref obj 0) "*TABLE*")
       (vector?  (vector-ref obj 1))))
```

– un *mutateur*, pour modifier un objet du type en affectant une valeur à un élément du tableau :

```
(define (table-set! T i j val)
  (if (table? T)
      (vector-set!
       (vector-ref
        (vector-ref T 1) i)
       j
       val)))
```

– un *sélecteur*, pour accéder à un élément d'un tableau du type :

```
(define (table-ref T i j)
  (if (table? T)
      (vector-ref
         (vector-ref
            (vector-ref T 1)
            i)
         j)))
```

– et diverses procédures utilitaires dont la fonction se comprend d'elle-même :

```
 (define (table-nlines T)
   (if (table? T)
       (vector-length (vector-ref T 1))))

(define (table-ncols T)
   (if (table? T)
       (vector-length (vector-ref (vector-ref T 1) 0))))

(define (table-print T)
   (if (table? T)
       (let ((n (table-nlines T))
             (m (table-ncols T)))
          (do ((i 0 (+ 1 i)))
              ((= i n))
              (let ((this-line
                     (vector-ref
                      (vector-ref T 1) i)))
                 (do ((j 0 (+ 1 j)))
                     ((= j m))
                     (display (vector-ref this-line j))
                     (display " "))
                 (newline))))))
```

Pour faire de cette collection de procédures un module compilable par Bigloo, on ajoutera en tête :

```
 (module tables
   (export
    (make-table n m)
    (table? obj)
    (table-set! T i j val)
    (table-ref T i j)
    (table-nlines T)
    (table-ncols T)
    (table-print T)))
```

Nous aurons ainsi créé um modules `tables`,
que nous pourrons utiliser, par exemple, dans le module
principal `use-tables` :

```
 (module use-tables
   (import tables)
   (main start))

(define (start args)
   (let ((i (string->integer (cadr args)))
  (j (string->integer (caddr args))))
      (build-table i j)))

(define (build-table i j)
   (let ((T (make-table i j)))
      (do ((k 0 (+ 1 k)))
   ((= i k))
   (do ((l 0 (+ 1 l)))
       ((= j l))
       (table-set! T k l (+ l k))))
      (table-print T)))
```