Chapitre 5 : Compléments sur les Types en OCaml en C

28 novembre 2023

1 Types Construits en OCaml

Les types construits (hors mutabilité) en OCaml incluent :

— Les **types produits** (n-uplets) : un n-uplet d'entiers, par exemple, est un élément de type `int * int * ... * int`. Les valeurs des n-uplets se notent entre parenthèses : le quadruplé d'entier contenant 4, 5, 6 et 7 se note `(4,5,6,7)`. Les éléments aux différentes positions d'un n-uplets peuvent être de types diférents : on peut par exemple avoir un couple de type **int * float**.

— Les **produits nommés** (ou enregistrements) : ce sont des types définis avec des champs fixés, où les champs prennent des valeurs d'un type défini. La syntaxe pour définir un type enregistrement est la suivante :

`type typeproduit = {champ1 : type1; ...; champn : typen } ;;`

Où l'on remplace `typeproduit` par le nom du type, `champ1,...,champn` par les noms de champs et `type1,...,typen` par les types associés aux champs.

La syntaxe pour affecter à une valeur à un élément de type produit est la suivante :

```
a = {champ1 = valeur1 ; ... ; champn = valeurn} ;;
```

— Les **énumérations** (ou sommes disjointes) où l'on définit un type qui peut prendre un ensemble de valeur parmi des **constructeurs** donnés. On définit un type somme en listant ses constructeurs (avec des majuscules), séparés par des barres verticales :

```
type typesomme = Constructeur1 | ... | Constructeurn ;;
```

On affectera simplement une valeur `Constructeur1`,...,`Constructeurn` à un élément de ce type.

— Les **sommes disjointes avec arguments** sont des sommes dans lesquelles les constructeurs peuvent (ou non) prendre un argument en entrée, de type défini. Pour définir un type somme où l'un des constructeurs prend en entrée un argument de type `type1`, on utilise la syntaxe suivante :

```
type sommearg = Constructeur1 | ... | Constructeurn of type1 ;;
```

Pour indiquer que l'on souhaite qu'une variable de ce type prenne la valeur `Constructeurn` avec en argument la valeur `valeur`, on utilise la syntaxe suivante :

```
a = Constructeurn valeur ;;
```

L'argument d'un constructeur peut être un élément du type en train d'être défini (ou construit à partir de ce type). Cela permet notamment de définir des types récursifs.

2 Types Mutables en OCaml

Les types mutables en OCaml incluent :

— Les **enregistrements modifiables**, dans lesquels les champs sont définis comme étant modifiables en plaçant le mot-clé `mutable` devant leurs noms dans la définition :

```
type enregistrementmod = {mutable champ1 : type1 ;
                          ... ;
                          mutable champn : typen} ;;
```

Pour un élément **e** de ce type, on modifie la valeur d'un champ mutable `champ` avec la syntaxe `champ.e <- valeur`.

— Les **références**, qui sont des objets de type `'a ref`. On affecte à une variable `x` la valeur d'une référence contenant la valeur `v` avec la syntaxe :

```
x = ref v ;;
```

La valeur `v` peut être récupérée avec l'expression `!x`. On peut modifier la valeur contenue dans `x` avec une instruction `x := v2`.

— Les **tableaux** sont des éléments de type `array`, qui contiennent une série de n éléments de même type stockés dans n cases et directement accessibles en mémoire. Un tableau peut être défini par la liste de ses éléments avec la syntaxe suivante :

```
t = [| e1 ; e2 ; ... ; en |] ;;
```

Les éléments sont indexés à partir de 0. On accède à l'élément d'indice `i` avec `t.(i)`.

3 Types Construits en C

3.1 Tableaux

Comme en OCaml, il est possible de définir en C des types construits, dont des tableaux.

En C, un **tableau** est un ensemble d'éléments de mêmes types, placés en mémoire à des adresses consécutives.

On définit un tableau avec la syntaxe suivante :

```
type nom_tableau[nombre_elements];
```

où `type` est le type des éléments du tableau, et `nombre_elements` une valeur **constante** correspondant au nombre de cases du tableau.

Comme en OCaml, les éléments dans le tableau sont indexés à partir de 0 : on accède au i-ème élément du tableau `t` avec la syntaxe `t[i]`.

Attention

La valeur correspondant à la taille du tableau doit être une **constante**, définie au moment de la compilation.

En général, on évite de donner un entier fixe prédéfini (10, 15, 20...) comme taille de tableau, noamment si on est amené à en créer plusieurs de même taille et que l'on n'est pas définitivement sûr de la valeur de la taille.

On peut définir `type tableau[N]` où l'on a défini `N` précédemment comme étant une constante, avec la syntaxe suivante :

```
const type N = valeur;
```

Exemple : pour fixer `N` à la valeur 10 :

```
const int N = 10;
```

(la syntaxe `#define` existe également mais est hors-programme)

En C, un tableau est en fait un **pointeur constant** vers l'adresse du premier élément (d'indice 0) : il n'est pas possible de lui affecter de nouvelle valeur simplement avec l'opérateur d'égalité.

En particulier, on ne peut pas définir un tableau comme étant égal à un autre avec la syntaxe `tab1 = tab2;`. Pour définir le contenu d'un tableau comme étant la copie du contenu d'un autre tableau, on doit utiliser une boucle pour modifier une à une les valeurs du premier tableau.

Pour initialiser les valeurs d'un tableau de taille `N` donnée, on peut utiliser la syntaxe suivante :

```
type t[N] = {constante1, ..., constanteN}
```

Attention : on ne peut pas utiliser cette syntaxe si `N` a été défini avec `const`.

3.1.1 Chaînes de Caractères

En C, une chaîne de caractère et simplement un tableau de type `char` : en effet, la construction en mémoire d'une chaîne de caractère est similaire à celle des tableau d'autres types avec les

pointeurs.

Une chaîne de caractère sera défini comme un tableau de caractères avec un dernier élément qui sera le caractère nul `'\0'`. Comme pour les tableau de type quelconque, on peut "lister les éléments" en écrivant simplement la chaîne de caractères :

```
char chaine[7] = "chaine"
```

Les éléments d'indice 0 à 5 sont ici les lettres du mot, l'élément d'indice 6 est l'élément nul.

Remarque : on peut également laisser vide le champ de la taille du tableau avec ce type de syntaxe, elle sera détectée automatiquement comme étant égale au nombre d'éléments du tableau.

La fonction `sizeof` permet de récupérer la place en mémoire d'un objet ou d'un type ; pour obtenir la taille d'un tableau en nombre d'éléments, on peut diviser la place en mémoire du tableau par la place en mémoire du type de ses éléments.

3.2 Tableau multidimensionnel

En OCaml, on peut définir des matrices à l'aide de tableaux de tableaux. De la même façon, on peut définir des tableaux multidimensionnels en C, avec la syntaxe suivante :

```
type nom_tableau[nombre_lignes][nombre_colonnes]
```

3.3 Structure

Une **structure** est une suite d'objets de types potentiellements différents, identifiés par des **champs** qui ont des noms. Ce type est analogue au **type produit** en OCaml.

La syntaxe pour définir un type structure est la suivante :

```
struct nom
{
type1 champ1;
type2 champ2;
...;
tyen champn;
}
```

Le type d'un objet comme défini précédemment sera `struct nom` : pour définir une variable `a` de ce tye, on utilisera donc la syntaxe :

```
struct nom a;
```

On accède à l'élément d'un champ donné (`champ1` par exemple) dans la variable avec la syntaxe `a.champ1`. Pour modifier la valeur du champ, on utilisera donc la syntaxe :

```
a.champ1 = valeur;
```

Contrairement au tableau, il est possible d'affecter à une structure la valeur d'une autre structure :

```
struct nom b = a;
```

3.3.1 `typedef`

Le fait de devoir noter `struct [nom]` pour définir chaque élément d'un type structure donné alourdit l'écriture.

Pour donner un autre identificateur à une structure, on peut utiliser la syntaxe suivante :

```
typedef [nom de structure] [nouvel identificateur];
```

Typiquement, on utilisera cela pour donner un nouveau nom au type structure, qui est le même que l'ancien débarrassé du mot-clé `struct` :

```
typedef struct nom nom;
```

Exemple. Définir un type `complexe` prenant deux champs flottants correspondant à la partie réelle et la partie imaginaire.

3.4 Enumération

Comme en OCaml, on peut définir des types `énumérations` qui prennent un ensemble fini de valeurs possibles dont on indique les noms. La syntaxe pour définir un type énumération prenant un ensemble de valeurs données est la suivante :

```
enum nom {valeur1; ...; valeurn};
```

Remarque. Les éléments sont représentés par des entiers en mémoire, de 0 pour la première valeur à $n-1$ pour la n-ième.

On peut en particulier définir le type booléen tel qu'il est construit en C de cette façon.

4 Pointeurs en C

4.1 Lvalues

En informatique, une **Left value** (ou Lvalue) est n'importe quel objet qui peut être placé à gauche d'un opérateur d'affectation.

Une Lvalue a :

— Une **adresse**

— Une **valeur**

L'adresse est un entier codé sur un certain nombre de bits (64 sur les ordinateurs récents).

Lorsque l'on définit une variable `v`, on accède à son adresse avec `&v`. Cette écriture a déjà été utilisée avec la fonction `scanf` pour permettre à l'utilisateur de rentrer le contenu d'une variable.

`&v` n'est pas modifiable : c'est une constante. on ne peut pas lui affecter de nouvelle valeur.

4.2 Pointeurs

Pour pouvoir manipuler des adresses qui soient modifiables, on utilise les pointeurs.

Définition

En C, les **pointeurs** sont des objets dont la valeur correspond à l'adresse d'un objet en mémoire. C'est une valeur modifiable.

La syntaxe pour définir un pointeur `p` vers une adresse contenu un objet de type `type` donné est la suivante :

```
type *p;
```

Le pointeur est un objet de type `type*` ; par exemple, un pointeur vers un entier est de type `int*`.

(L'intérêt de connaître le type de l'objet pointé est de définir l'espace en mémoire à lire pour connaître la valeur de l'objet)

La valeur pointée par un pointeur `p` est accessible avec la notation `*p`.

Les deux valeurs `p` et `*p` sont modifiables. On peut par exemple :

— Faire pointer `p` vers une variable `i` avec `p = &i;`

— Modifier le contenu de `i` à partir de là, avec l'instruction `*p = valeur;`

4.3 Opérations sur les Pointeurs

Un pointeur est un nombre entier, ce qui permet de faire des opérations arithmétiques sur ceux-ci. En particulier, on peut :

— **additionner** un entier à un pointeur (résultat : pointeur du même type que le pointeur initial)

— **soustraire** un entier à un pointeur (idem)

— calculer la **différence** entre deux pointeurs (résultat : entier)

L'opération d'addition de i à un pointeur consiste à se déplacer de i fois l'espace en mémoire du type donné.

4.4 Pointeurs et Tableaux

En C, **un tableau est un pointeur constant :** sa valeur est l'adresse du premier élément du tableau.

l'expression `tab` correspond à l'expression `&tab[0]` ; l'expression `*tab` correspond à l'expression `tab[0]`.

De même, l'adresse du i-ième élément du tableau est $p + i$, avec l'opération d'addition telle que définie sur les pointeurs.

`tab[1]` correspond à `*(tab+1)`, `tab[2]` correspond à `*(tab+2)`... Plus généralement, `tab[i]` correspond à `*(tab+i)`.

C'est également vrai pour les chaînes de caractères : une chaîne de caractère est un tableau de caractères, c'est donc un objet de type `char*`.

Pour les tableaux à n dimensions : un tableau à 2 dimensions est un tableau de tableaux. C'est donc un pointeur vers des éléments de type tableaux, donc un pointeur vers un élément de type `type*`.

Par conséquent, les tableaux en deux dimensions sont des éléments de type `type**` (par exemple `int**` pour des tableaux d'entiers à 2 dimensions).

4.5 Pointeur `NULL`

Il existe un pointeur prédéfini qui ne pointe pas vers un emplacement mémoire valide : c'est le **pointeur nul**, noté `NULL`. Il peut être utilisé pour initialiser la valeur d'un pointeur dont on ne connaît pas encore l'adresse, ou encore pour la construction de structures chaînées (cf chapitre suivant). On peut tester le fait qu'un pointeur `p` soit `NULL` ou non avec `p == NULL`.

On ne peut pas déréférencer un tel pointeur : une tentative de déréférencement d'un pointeur qui vaut `NULL` entraîne une erreur de segmentation.

Attention

Le langage C n'empêche pas, techniquement, de créer des pointeurs et de demander à les déréférencer à un moment où celui-ci serait accidentellement `NULL`, c'est-à-dire dont la zone mémoire serait désallouée. Il faut donc faire attention à ce que ce ne soit pas le cas.

Exemple. Soit fonction `f` contenant une déclaration de variable `v` et affectation de valeur `int v = 42;`, et finit par `return &v;`. La variable `v` cesse d'exister à la fin de l'exécution de `f` et la valeur renvoyée est un **pointeur fantôme**.

4.6 Allocation Dynamique

On rappelle que pour un pointeur `p`, il est possible à la fois de modifier `p` (adresse contenue dans la variable) et `*p` (valeur contenue à cette adresse).

Lorsque l'on crée un pointeur `p`, on cherche à initialiser son adresse mémoire. On peut initialiser par `p == NULL` si on ne connaît pas encore celle-ci. Une autre façon de faire est de l'initialiser par l'adresse d'une variable existante : `p = &var;`.

On peut également faire en sorte que `p` ne pointe pas vers une variable existante, et travailler directement avec la valeur de `*p`, en réservant un espace mémoire de taille adéquate. La réservation d'espace mémoire pour l'objet pointé se nomme **allocation dynamique** et se fait avec la fonction `malloc` de la libraire `<stdlib.h>`, dont la signature est :

```
void* malloc(size_t nb_octets)
```

où `size_t` est un type entier pour la taille en mémoire. On peut appeler la fonction `sizeof` sur un type pour obtenir une valeur de `nb_octets` :

```
int *p = malloc(sizeof(int))
```

Cet appel va affecter à `p` l'adresse d'un octet en mémoire, et pour le nombre n d'octets qui constituent le type `int`, va placer la valeur 0 dans les n octets consécutifs en mémoire à parir de l'adresse de `p`.

`malloc` est de type `void*` car les résultats renvoyés peuvent être des adresses de types différents selon la situation : `float*`, `int*`...

L'intérêt de la fonction `sizeof` est la **portabilité** du code : lorsque l'on définit le type d'un pointeur, on définit l'arithmétique dessus. Si un élément de type `int` est codé sur 4 octets, l'utilisation d'espaces voisins pour stocker plusieurs éléments de ce type (dans un tableau par exemple) se fait de 4 octets en 4 octets : la différence entre `p+1` et `p` sera de 4 octets, entre `p+2` et `p` sera de 8 octets, etc.

Or la taille de l'espace en mémoire utilisé pour stocker une variable d'un type donné dépend du compilateur : ainsi, utiliser `sizeof(int)` plutôt que 4 comme paramètre dans `malloc` permet de s'assurer que l'on n'aura pas d'incohérences au moment de la compilation.

La fonction `malloc` peut également être utilisée pour réserver plusieurs espaces contigus en mémoire, de même type, en demander d'allouer n fois la taille d'un type donné :

```
int *p = malloc(10*sizeof(int))
```

La ligne précédente permet d'allouer l'espace mémoire pour `p`, mais aussi, comme on l'a dit précédemment, pour `p+1`, `p+2`, ... et `p+9`.

On termine l'utilisation du pointeur `p` en **libérant** l'espace mémoire, avec la commande :

```
free(p);
```

Attention

A toute allocation en mémoire avec `malloc` doit être associée une libération de l'espace avec `free` pour éviter le risque de **fuites de mémoires**.

4.7 Pointeurs et Fonctions

4.7.1 Passage par Valeur, Passage par Adresse

On rappelle que le langage C utilise le **passage par valeur** : lorsque l'on définit une fonction `f` d'argument `i` et que `v` est donné en paramètre de la fonction, `f` copie la valeur de `v` dans une variable `i`, et agit sur `i` sans toucher à `v`. De cette façon, une fonction ne peut pas *a priori* modifier la valeur en mémoire de la variable en entrée.

Exemple. On considère la fonction suivante :

```
void incr(int i){
    i = i+1;
}
```

Et la séquence de commandes suivante appelant la fonction :

```
int v = 42;
incr(v);
```

Lors de l'exécution de cette fonction :

- La valeur 42 de `v` est copiée dans une nouvelle variable `i` ;
- La valeur de `i` augmente d'une unité (et pas celle de `v`) ;
- `i` disparaît à la fin de l'exécution de la fonction, et `v` reste inchangée.

Avec les pointeurs, on peut utiliser le **passage par adresse** : on fait en sorte qu'une fonction ne prenne plus en entrée comme paramètre la valeur de la variable `v`, mais la valeur de son adresse, c'est-à-dire `&v`. La variable créée par le programme sera donc un pointeur vers l'adresse de `v` et permettra de modifier son contenu.

Exemple. On modifie la fonction précédente pour prendre en entrée un pointeur vers un entier :

```
void incr(int *i){
    *i = *i+1;
}
```

Et la séquence de commandes suivante appelant la fonction :

```
int v = 42;
incr(&v);
```

Lors de l'exécution de cette fonction :

- La valeur de l'adresse de `v` est copiée dans une nouvelle variable `i` : `*i` et `v` correspondent au **même espace en mémoire.**
- La valeur de `*i` - donc celle de `v` - augmente d'une unité ;
- `i` disparaît à la fin de l'exécution de la fonction, et `v` a été modifiée.

4.7.2 Renvoi de Plusieurs Valeurs

Imaginons que l'on souhaite écrire une fonction de division euclidienne qui prenne en entrée deux entiers a et b et renvoie le quotient q et le reste r de la division euclidienne de a par b. Cela est faisable simplement en OCaml en renvoyant un couple de valeurs.

En C, le type n-uplet n'existe pas, et une fonction ne renvoie qu'un élément : on ne peut donc pas procéder ainsi. Par ailleurs, l'utilisation de structures implique de créer un (ou plusieurs) type structure avant, ce qui peut être superflu si son nombre d'utilisations est limité.

Une alternative est de renvoyer une valeur et de prendre en argument de la fonction un (ou plusieurs) pointeurs, et écrire les autres valeurs dans des variables dont on fournit l'adresse comme paramètre à la position de ses arguments.

Exemple. Pour la division euclidienne, on peut écrire une fonction de signature :

```
int division(int a, int b, int *r)
```

Et créer deux variables `q` et `r` qui contiendront le quotient et le reste de la division euclidienne de 547 par 32 avec les deux lignes suivantes :

```
int r;
int q = division(547,32,&r);
```

4.8 Pointeurs et Structures

En C, on utilise fréquemment des pointeurs pour les fonctions prenant en entrée des structures comme paramètres.

En effet, on rappelle qu'en C, le passage par valeur fait que l'appelle d'une fonction entraîne la création d'une copie de cet élément dans une nouvelle variable qui sera modifiée au cours de l'exécution de la fonction. Les structures pouvant être coûteuses en place, créer des copies de structures entraîne une occupation importante de l'espace mémoire.

Par conséquent, lorsque l'on souhaitera agir sur les champs des structures de type `struct S` - pour les modifier ou pour simplement récupérer leurs valeurs - on prendra en entrée un élément de type `struct S *`. Pour une variable `x` de ce type, l'accès à des champs `c1` ou `c2` se fera par `(*x).c1` ou `(*x).c2`.