# Type option en OCaml



## I Principe

Dans de très nombreux contextes, on a besoin de pouvoir exprimer une notion de partialité en programmant. Cela apparait en général dans deux cas :

- On veut réaliser une fonction qui ne va pas pouvoir renvoyer une valeur dans tous les cas. Exemple : renvoyer la tête d'une liste chaînée ne fonctionne pas si on passe la liste vide. La fonction est en fait une fonction partielle au sens mathématiques
- On souhaite construire progressivement une donnée et il faut qu'on puisse avoir une notion de valeurs indéterminées. Exemple : on veut remplir au fur et à mesure une grille de Sudoku, à la fin, c'est une matrice de nombres, mais il faut pouvoir gérer les cases vides de manière intermédiaires.

En C, on résout ces questions en utilisant des valeurs comme `-1` ou le pointeur nul `NULL`. En `OCaml`, le système de types nous pousse à chercher une meilleure solution.

## II Syntaxe

La solution en `OCaml` est très simple, mais demande de la pratique pour l'utiliser à bon escient. Il s'agit, pour tout type `'a` de définir un type `'a option` permettant de représenter, soit une valeur de type `'a`, soit une absence de valeur.

Ce type est défini ainsi :

```OCaml
type 'a option = None | Some of 'a
```

Cela signifie qu'on a remplacé les valeurs `b` par des valeurs `Some b` et que l'absence de valeur est maintenant une vraie valeur `None`.

Aucune difficulté pour définir une valeur de type option, on se contente d'appeler un des deux constructeurs. Ainsi `Some 1` est un `int option` et `Some "test"` un `string option`. La valeur `None` est polymorphe dans le même sens que `[]`.

Pour manipuler une valeur `o` du type `'a option`, on effectue un filtrage comme pour les autres types somme :

```OCaml
match o with
| None -> (* ... *)
| Some a -> (* ici on peut accèder au contenu a *)
```

Il est également possible de revenir sur le comportement précédent avec une fonction comme

```OCaml
let unwrap o =
    match o with
    | None -> failwith "None"
    | Some a -> a
```

Qui permet de déballer une valeur `'a option` en faisant l'hypothèse que ce n'est pas `None`.

**Remarque** Une erreur classique avec le type option, qui est la même qu'avec le type `list`, c'est de se focaliser sur `None` en commençant par écrire :

```OCaml
if o = None
then (* .. *)
else (* et ici on est bloqué, car o est emballé *)
```

Donc, une règle : si on a une valeur `option`, on effectue un filtrage !

# III Utilisation concrète

## III.1 Définir une fonction partielle

Il se trouve qu'on a déjà vu un mécanisme permettant de définir une fonction qui peut échouer : les exceptions, notamment avec `failwith`. Si on considère une fonction `f : 'a -> 'b` on a ainsi deux types de valeurs dans le type `'a` :

- Les valeurs `x` qui permettent d'obtenir une valeur `f x` du type `'b`
- Les valeurs `y` pour lesquelles `f y` produit une erreur.

Schématiquement, si note $A$ les valeurs du type `'a`, on a donc $A = A_s \cup A_e$ où l'union est disjointe et $A_s$ sont les valeurs pour lesquelles `f` ne produit par d'erreurs et $A_e$ les valeurs produisant des erreurs. Si on prend la définition usuelle de fonctions en mathématiques, $A_s$ est le domaine de `f` et on considère en fait une **application** de $A_s$ dans $B$ les valeurs du type `'b`.

L'idée avec le type `option`, c'est de transformer toute fonction partielle $f : A \to B$ en une application $f^\star : A \to B \cup \{\star\}$ où $\star$ est un élément spécial $\star \notin B$ qui correspond à une **valeur** erreur. Cela correspond à une *réification* de l'erreur au rang de valeur. L'application $f^\star$ est ainsi définie :

$$\begin{array}{rccl} f^\star & : & A & \to & B \cup \{\star\} \\ & & a & \mapsto & \begin{cases} f(a) & \text{si } a \in A_s \\ \star & \text{si } a \in A_e \end{cases} \end{array}$$

La valeur $\star$ est ce qu'on appelle un *puit*, c'est ici qu'on redirige toutes les entrées invalides.

Il n'y a pas de type correspondant directement à l'union $B \cup \{\star\}$ en `OCaml` et c'est pour cela qu'on utilise le type `'b option` qui nous force à *emballer* un retour.

Ainsi à chaque fois qu'on aurait renvoyé une valeur `b`, on va renvoyer plutôt une valeur `Some b`, et à la place de produire une erreur, on renvoie `None`.

Par exemple, le code suivant :

```OCaml
let tete l =
    match l with
    | t::q -> t
    | [] -> failwith "Liste vide"
```

va s'écrire

```OCaml
let tete_opt l =
    match l with
    | t::q -> Some t
    | [] -> None
```

## III.2 Appeler une fonction partielle

Pour appeler une fonction partielle, il n'y a pas de difficulté, il suffit de faire l'appel de fonction et de manipuler ensuite le type option comme vu plus haut. Cela signifie qu'on effectuera en général l'appel directement dans l'expression d'un filtrage :

```OCaml
match f a with
| None -> (* ... *)
| Some b -> (* ... *)
```

## III.3 Données partielles

Pour tout type polymorphe `'a t` on peut en déduire un type partiel `'a option t` dans lequel les valeurs peuvent ne pas être définies. Ainsi, un `int option array` permettra pour chaque case du tableau de :

- soit être définie, et ce sera alors une valeur de la forme `Some k` où `k` est un `int`,
- soit être indéfinie avec la valeur `None`.

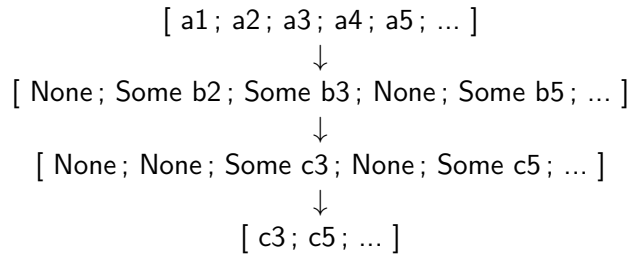
Ainsi, une grille de Sudoku qu'on remplirait progressivement aurait le type `int option array array`. Une case valant `None` signifiant qu'elle n'est pas encore remplie. Bien entendu, une fois la grille remplie, toutes les valeurs seraient de la forme `Some k`. On peut ainsi imaginer une transformation de `int option array array` vers `int array array` consistant à passer d'une grille pouvant être partielle, mais pleinement remplie à une grille d'entiers :

```OCaml
let grille_complete go =
    let g = Array.make_matrix 9 9 0 in
    for i = 0 to 8 do
        for j = 0 to 8 do
            match go.(i).(j) with
            | None -> failwith "Grille incomplète"
            | Some k -> g.(i).(j) <- k
        done
    done;
    g
```

## III.4 Chaîne de traitement

On pourrait se dire que le type `option` n'est qu'en fait une version un peu lourde de la production d'erreurs. Mais en fait, la force du type `option` est justement que c'est un type comme un autre. Cela permet d'effectuer de nombreux traitements en manipulant des fonctions partielles pour définir des données partielles et d'attendre uniquement au dernier moment pour déballer les valeurs. L'idée forte est de manipuler la partialité naturellement.

Un exemple cela peut-être d'avoir une série de traitement à appliquer sur une liste, chaque traitement va possiblement faire apparaitre des `None` qu'on va traiter naturellement avec éventuellement une étape finale pour retirer les options :




On remarque que ce style de programmation est de plus en plus privilégié par la bibliothèque standard de `OCaml` :

- de nombreuses fonctions partielles `f : 'a -> b'` ont une variante `f_opt : 'a -> 'b option`
- il y a des fonctions manipulant directement des `_opt` comme `List.filter_map` qui effectue un `map` avec une fonction `'a -> 'b option` et filtre les `None` en sortie.
- Le nouveau module `Option` contient beaucoup de fonctions pour se simplifier la vie quand on programme avec les types `option`. **Attention**, cela correspond à une utilisation bien plus pointue de `OCaml` que dans le reste de ce cours. Notamment, cela fait beaucoup de sens en lien avec l'opérateur `|>` qu'on n'utilise pas ici.