Chapitre I

Définitions et notations

Les solutions des exercices proposés dans ce livre sont écrites en OCaml. Sans rentrer dans une présentation exhaustive de ce langage – qui n'est pas le propos de cet ouvrage – nous donnons dans ce chapitre les principales notations utilisées dans la suite pour présenter les solutions des exercices.

L'un des objectifs de cet ouvrage est d'associer à chacune des fonctions OCaml que nous proposons une preuve de terminaison et de correction. Nous rappelons donc dans ce chapitre la technique de *preuve par induction*, qui sera utilisée pour prouver les solutions rigoureusement.

Enfin, nous complétons également la définition de chaque fonction par une analyse de sa *complexité*. Nous rappelons donc les principales hypothèses liées à ce type d'analyse, et nous précisons celles que nous avons retenues dans la suite pour présenter ces résultats de complexité.

1 Notations et rappels en OCaml

Nous présentons tout d'abord les opérateurs arithmétiques portant sur les entiers et les nombres flottants, sachant que nous utilisons principalement les entiers, à l'exception de l'Exercice 8. Cet exercice fournira ainsi l'occasion de signaler la différence entre les nombres réels et leur représentation en machine. Nous rappelons ensuite les notations OCaml pour noter les fonctions puis pour représenter les listes, et enfin nous définissons les structures arborescentes.

1.1 Types et opérations de base

Nous utilisons le type booléen noté `bool` et les opérations « *et* » noté `&&` et « *ou* » noté `||`, mais aussi les types entier (`int`) et réel (`float`) de OCaml. Les opérations usuelles manipulant ces types sont notées comme suit :

- `+,-,*,/, mod` représentent respectivement l'addition, la soustraction, la multiplication, le quotient et le reste de la division entière.
- `+.,-.,*.,/.` représentent respectivement l'addition, la soustraction, la multiplication, la division de nombres flottants. De plus, les nombres flottants sont toujours écrits avec le « `.` » séparant la partie entière de la partie décimale, même lorsque celle-ci est vide. Ainsi, l'expression « `2.` » dénote la valeur réelle 2.

1.2 Fonctions

Les solutions des exercices sont toutes décrites par une (ou plusieurs) fonction(s) OCaml. Nous n'utilisons que la partie dite *fonctionnelle* du langage : les fonctions renvoient toutes un *résultat* qui est obtenu par *composition* d'autres fonctions ou opérateurs du langage, à partir de *paramètres*. Ces fonctions seront souvent *récursives*, reconnaissables par le mot clef *rec* du langage OCaml.

Le type d'une fonction est défini à partir du type de ses paramètres et du type de son résultat. Dans le cas de fonction à plusieurs paramètres nous utilisons la forme dite « currifiée », qui permet l'application de fonctions partielles. Ainsi, le type d'une fonction prenant deux paramètres, un entier et un booléen, et renvoyant un réel aura pour type `int -> bool -> float`.

Comme certains opérateurs prédéfinis, une fonction peut également être polymorphe dans le sens où son résultat et/ou certains de ses paramètres peuvent être de différents types. En OCaml un type polymorphe α, β, etc s'exprime par une variable de type notée `'a`, `'b`, etc. La fonction **let** id x = x implémente la fonction *identité* de type `'a -> 'a`. De même, le type de l'opérateur polymorphe « `=` », qui permet de comparer les valeurs de deux éléments d'un même type quelconque, est `'a -> 'a -> bool`. Ainsi par exemple, le test d'égalité entre deux entiers, deux réels, deux listes ou plus généralement deux éléments de type quelconque est notée avec l'opérateur « `=` ». Les expressions `2 = 2`, `2. = 2.` ou `[1;2;3] = [1;2;3]` ont pour valeur `- : bool = true` en OCaml.

Enfin, l'une des caractéristiques des langages fonctionnels comme OCaml est de considérer que les fonctions sont des types de données comme les autres, dans le sens où une fonction peut être transmise en paramètre à une fonction, ou renvoyée comme résultat par une fonction. Dans ce cas les fonctions sont dites d'*ordre supérieur*.

1.3 Types et opérations sur les couples

Lorsqu'une fonction donne plusieurs résultats ceux-ci sont rendus dans une structure de données. Nous n'aurons besoin dans cet ouvrage que de fonctions

qui donnent deux résultats, `a` et `b`. Ceux-ci sont alors rendus dans un couple noté : `(a,b)`.

L'accès aux composants d'un couple est possible avec les opérateurs de sélection : `fst` et `snd` ou en utilisant la construction `let in`. Par exemple soit une fonction `fc : 'a -> 'b * 'c`. Nous écrivons `let (a,b) = fc (e) in a` pour obtenir la première composante du couple `(a,b)` issu du calcul de la fonction `fc` appliquée à `e` .

1.4 Types et opérations sur les listes

Les listes sont modélisées en OCaml par deux *constructeurs* : la liste vide, notée `[]`, et l'opérateur `::`, permettant d'ajouter un élément à gauche d'une liste[1]. A ces constructeurs correspondent les opérateurs de sélection (ou *sélecteurs*) suivants, définis dans le module `List` qui fait partie des bibliothèques standards de OCaml [Leroy and Weis, 1993] :

- `List.hd`, pour *head*, qui prend en paramètre une liste non vide et renvoie l'élément de tête de cette liste ;
- `List.tl`, pour *tail*, qui prend en paramètre une liste non vide et renvoie cette même liste privée de son premier élément.

Ce module contient de nombreuses autres fonctions sur les listes fréquemment utilisées en OCaml. Certains exercices consistent à réaliser quelques unes de ces fonctions comme `List.map` ou encore `List.find`, ceci afin de mieux en comprendre l'essence et de pouvoir les manipuler avec plus d'aisance.

En OCaml, les listes sont polymorphes. Par contre tous leurs éléments sont d'un (même) type quelconque `'a`. Le type OCaml correspondant à une liste est `'a list`. Par suite, le type du constructeur `[]` est `'a list`, celui du constructeur `::` est : `'a -> 'a list -> 'a list`. Par exemple, la liste des trois premiers entiers positifs de type `int list` est construite par `1 :: (2 :: (3 :: []))` ; elle est aussi notée `[1;2;3]`.

Pour traiter des listes *non vides* d'entiers (par exemple dans la Solution 20), nous utilisons le type OCaml récursif suivant :

```
type listenonvide =   Elem of int
                    | Cons of int * listenonvide
```

Ce type représente une structure de donnée identique à celle des listes en OCaml, sauf que toutes les listes sont non vides. Par exemple, la fonction récursive `f`, qui parcourt une liste non vide et la re-construit à l'identique s'écrit comme suit :

```
let rec f liste = match liste with
```

1. Nous dirons aussi ajouter un élément en tête de la liste.

```
  | Elem(x) --> Elem(x) (* liste ayant un seul élément *)
  | Cons(p, fin) --> Cons(p, f fin)      (* cas général *)
```

Il est aussi possible d'utiliser le type `list` du langage OCaml à condition de gérer l'erreur en cas de liste vide comme le montre le code de la fonction récursive `f2`. Cela nécessite d'utiliser par exemple la fonction `failwith` qui prend en paramètre une chaîne de caractères et qui lève une exception ayant pour nom cette chaîne de caractères.

```
let rec f2 liste = match liste with
  | [] --> failwith "fonction non definie"
  | [x] --> [x]          (* liste ayant un seul élément *)
  | p::fin --> p::(f2 fin)              (* cas général *)
```

1.5 Types et opérations sur les arbres

La notion d'*arbre* (ou plus généralement d'*arborescence*) est à la base de structures de données largement utilisées en informatique. Nous nous intéressons uniquement aux *arbres binaires*, étiquetés par des éléments de type quelconque, définis en OCaml par le type récursif suivant :

```
type 'a arbre = V | N of 'a arbre * 'a  *  'a arbre
```

Intuitivement, le constructeur `V` représente un *arbre vide* et le constructeur `N(g, e, d)` représente un arbre dont le sous-arbre gauche (ou *fils gauche*) est l'arbre `g`, le sous-arbre droit (ou *fils droit*) est l'arbre `d`, et la *racine* est un nœud dont l'étiquette associée est `e`. Notons qu'une *feuille* est un nœud qui a l'arbre vide `V` pour fils gauche et droit. Enfin, la *hauteur* d'un arbre binaire est la longueur du plus long chemin (en nombre de nœuds) allant de la racine à une feuille de cet arbre. Par définition un arbre vide `V` est de hauteur nulle.

Ainsi l'arbre binaire d'entiers `N( V, 1, N ( N ( V, 3, V ), 2, V ) )` correspond à l'arbre de la Figure I.1. Le nœud d'étiquette `1` est la racine, celui d'étiquette `3` est la seule feuille de cet arbre de hauteur 3.

Nous avons choisi ici d'étiqueter les nœuds de nos arbres binaires. Il existe de nombreuses autres modélisations de structures arborescentes comme par exemple les arbres n-aires, les arbres équilibrés, les arbres Rouge-Noir, ou encore les arbres AVL introduits par Adelson Velskii et Landis en 1962.

2 Preuve de terminaison et de correction

Nous donnons ici les éléments utilisés dans la suite du livre pour aborder les questions de terminaison et de correction d'une fonction (récursive).

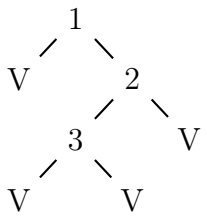

FIGURE I.1 – Exemple d'arbre binaire

2.1 Terminaison d'une fonction récursive

La terminaison d'une fonction récursive nécessite d'associer à la fonction une *mesure* définie sur un ensemble E muni d'un ordre bien fondé [2] ($\prec$). Dans la plupart de nos exercices l'ensemble E sera l'ensemble des entiers positifs, muni de la relation d'ordre usuelle ($<$). Dans certains cas, il est nécessaire de définir un ordre sur des couples d'entiers appelé *ordre lexicographique* que nous notons $<_{lex}$: $(x_1, y_1) <_{lex} (x_2, y_2) \iff (x_1 < x_2) \vee ((x_1 = x_2) \wedge (y_1 < y_2))$.

Pour prouver la terminaison d'une fonction récursive `f`, il suffit alors de montrer que :

- la fonction termine pour chacun des *cas de base*(un cas de base est un cas pour lequel il n'y a pas d'appels récursifs), la mesure associée à chacun de ces cas de base est donc un élément m_0 de E ;
- la mesure obtenue pour chaque appel récursif est strictement inférieure à celle associée à l'appel courant, et la séquence des mesures $[m_n, m_{n-1}, \ldots]$ correspondant à une suite d'appels récursifs converge vers au moins une des valeurs m_0.

À titre d'exemple, considérons la fonction récursive `double_s` de type `int -> int`, qui prend un entier, renvoie son double et est définie sur les entiers positifs comme suit :

```
let rec double_s n =
    if n < 0 then failwith "fonction non definie"
    else if n = 0 then 0 else 2 + (double_s (n - 1));;
```

Choisissons comme mesure la valeur du paramètre `n`, de type entier positif. Le cas de base, `double_s 0` termine (il vaut 0), et la mesure m_0 vaut donc ici 0. Lors d'un appel récursif de `double_s n`, le paramètre devient `n - 1` donc la mesure associée décroît également de une unité. Par suite, la séquence des mesures obtenues convergera vers la valeur m_0, ce qui assure la terminaison de la fonction. Remarquons au passage que cette fonction ne terminerait pas si le cas où le paramètre transmis est négatif n'était pas pris en compte par un

2. c'est-à-dire dans lequel toute séquence strictement décroissante est finie.

`failwith`.

Dans le cas de fonctions portant sur des listes, il est souvent pertinent de choisir comme mesure la *taille* (*i.e.,* le nombre d'éléments) d'une (ou plusieurs) des liste(s) passées en paramètre. Par exemple, pour la fonction `lg` calculant la longueur d'une liste (*c.f.* section 3.4), le cas de base termine et il porte sur la liste vide donc m_0 vaut 0. De plus, chaque appel récursif porte sur une liste non vide privée de son premier élément, la mesure associée décroît donc strictement de une unité, la séquence des mesures obtenues convergera alors vers la valeur m_0.

Enfin, dans le cas de fonction portant sur des arbres, la mesure considérée pourra être le nombre de nœuds de l'arbre, ou encore sa hauteur.

2.2 Correction et raisonnement par induction

Les preuves que nous développons font appel à des calculs arithmétiques, mais elles sont également le plus souvent basées sur un raisonnement par *récurrence/induction*, qui permet de montrer qu'une propriété P est vraie pour tous les éléments d'un ensemble (fini ou non) muni d'un ordre bien fondé. Nous illustrons ici cette technique en considérant l'ensemble des entiers positifs muni de la relation d'ordre $<$.

Nous rappelons tout d'abord le principe de la *récurrence*. Pour montrer qu'une propriété P portant sur un entier n positif est vraie quel que soit n, nous montrons successivement que :

1. la fonction est correcte pour les cas de base, par exemple $P(0)$ est vrai ;
2. si nous supposons la fonction correcte pour un certain entier $n \geq 0$ (*hypothèse de récurrence*), alors celle-ci est encore correcte pour l'entier $n+1$:
$$\forall n \geq 0,\ P(n) \implies P(n+1)$$

Dans le cas d'une *induction généralisée*, la seconde étape consiste à supposer la propriété vraie *pour tout* entier k compris entre 0 et n (*hypothèse d'induction*), et montrer qu'elle est alors encore vraie pour l'entier $k+1$. Plus formellement, nous montrons :

$$\forall n \geq 0,\ P(1) \wedge \ldots \wedge P(n) \implies P(n+1)$$

À titre d'exemple nous prouvons par récurrence sur n que pour tout $n > 0$ la fonction `double_s n` calcule $2 \times n$, c'est-à-dire le double de n.
Preuve :

– cas `n = 0` : la définition de la fonction donne `double_s 0 = 0`, qui est le double de 0.

– récurrence : supposons que la fonction `double_s k` calcule le double de `k`, pour tout `k` positif. Montrons alors que `double_s k+1` calcule le double de `k+1`. D'après la définition de la fonction `double_s k+1` vaut `2 + double_s k`. Par hypothèse de récurrence `double_s k` vaut $2 \times k$. Ainsi la fonction `double_s k+1` vaut $2 + 2 \times k = 2 \times (k + 1)$ qui est le double de `k + 1`.

$CQFD$

3 Complexité d'une fonction

Analyser la complexité d'une fonction consiste à évaluer certains paramètres liés à son exécution, comme le temps et la quantité de mémoire nécessaire. Ces valeurs dépendent en général d'une part des paramètres transmis à cette fonction, et, d'autre part, de la plateforme d'exécution considérée (processeur, système d'exploitation, etc.). Pour obtenir des résultats pertinents dans le cas général, il est donc nécessaire d'effectuer un certain nombre d'hypothèses.

3.1 Valeurs des paramètres et fonctions de coûts

En premier lieu, le comportement d'une fonction à l'exécution, et donc le nombre d'opérations qu'elle va effectuer ou l'espace mémoire qu'elle va occuper, dépendent en général fortement des valeurs des paramètres sur lesquels elle opère. Par exemple, le temps d'exécution d'une fonction effectuant une recherche séquentielle d'un élément dans une liste de grande taille pourra être très différent selon que cet élément se trouve en *tête* ou en *queue* de la liste.

Par suite, pour donner une mesure de complexité, il est nécessaire de faire un certain nombre d'hypothèses sur les valeurs de ces paramètres. Dans ce livre, nous choisissons de nous intéresser à la complexité "au pire cas", qui consiste à considérer les *valeurs des paramètres les plus défavorables* (du point de vue de la complexité), typiquement en supposant que l'élément cherché est en queue de liste, dans l'exemple précédent. Cette solution permet notamment de s'affranchir d'hypothèses probabilistes sur les valeurs de ces paramètres, ce qui serait le cas pour une analyse "en moyenne". Par suite, la complexité d'une fonction s'exprime souvent indépendamment de la valeur exacte de ses paramètres, mais plus simplement en fonction de la "taille" de leur représentation en machine. Cela est notamment le cas des fonctions portant sur des listes (*c.f.* section 3.4) ou sur des arbres (*c.f.* section 3.5), pour lesquelles le critère principal de complexité est le nombre d'éléments dans la liste, ou le nombre de nœuds dans l'arbre. Nous raisonnons ainsi sur une *partition* des valeurs des

paramètres : la complexité de la fonction est identique pour toutes les listes (ou arbres) *de même taille* quelles que soient les valeurs des éléments composant ces listes ou arbres.

Il en résulte que la complexité d'une fonction s'exprimera par des *fonctions de coûts*, qui associent à chaque valeur (ou partition sur les valeurs) des paramètres une mesure de leur temps d'exécution ou de leur consommation en mémoire.

Remarquons enfin que les travaux portant sur la théorie de la complexité des algorithmes s'intéressent rarement aux calculs des coûts exacts mais effectuent une *analyse asymptotique*. Il s'agit ici de s'affranchir des constantes qui apparaîssent dans les fonctions de coûts et de ne considérer que leur *ordre de grandeur*, obtenu pour de très grandes valeurs des paramètres. Pour ce faire, il est utile de savoir encadrer une fonction de coût. Par exemple, les fonctions $n \mapsto 2 \times n$ ou $n \mapsto 10 \times n$ ont le même *comportement asymptotique* que la fonction $n \mapsto n$. Nous indiquons ici les trois manières principales de formaliser cette idée, en introduisant les notations utilisées classiquement :

$$\begin{aligned}
f(n) \in O(g(n)) &\Leftrightarrow \exists k > 0, \exists n_0\, \forall n > n_0, |f(n)| \leq |g(n)| \times k \\
f(n) \in \Omega(g(n)) &\Leftrightarrow \exists k > 0, \exists n_0, \forall n > n_0\ |g(n)| \times k \leq |f(n)| \\
f(n) \in \Theta(g(n)) &\Leftrightarrow \exists k_1, k_2 > 0, \exists n_0, \forall n > n_0 \\
&\qquad |g(n)| \times k_1 < |f(n)| < |g(n)| \times k_2
\end{aligned}$$

Précisons que la notation grand O correspond à une majoration de la complexité, la notation grand oméga Ω à une minoration et grand théta Θ à un encadrement. Il existe des définitions similaires avec les inégalités strictes, notées par les mêmes lettres en minuscule.

3.2 Plateforme d'exécution et coût des opérations de base

Une seconde hypothèse nécessaire concerne la plateforme sur laquelle la fonction est exécutée. En effet, aussi bien les temps d'exécution que les espaces mémoires utilisés dépendent de cette plateforme. Là encore, pour obtenir des résultats de complexité suffisamment généraux, l'approche usuelle consiste à considérer un *modèle* de cette plateforme, dans lequel nous choisissons un certain nombre d'opérations élémentaires supposées représentatives du temps d'exécution et de la taille mémoire.

Dans notre modèle, le temps d'exécution des opérations élémentaires portant sur des entiers, sera supposé *constant* et indépendant de la valeur de ces entiers. Nous considérons de plus que tous les entiers sont représentés en machine sur une taille mémoire unique noté s_{int}. Ainsi le coût en temps de la multiplication (opérateur $*$) entre deux entiers est la constante notée $c^*(int)$.