Précis de programmation fonctionnelle à l'usage des débutants en informatique et de ceux qui croient ne plus l'être

Hervé Frezza-Buet

Table des matières

Chapitre 1

Introduction

1.1 Ambitions

Le présent document n'a pas pour vocation d'être exhaustif sur la programmation fonctionnelle. Il ne remplace donc pas de bons ouvrages de référence, comme [Paulson, 1991] ou [Wikström, 1987]. L'objectif ici est seulement de présenter une introduction pratique à la programmation fonctionnelle, avec peu de références à la théorie de la réécriture qui lui est sous-jacente. Cette introduction pose par des exemples quelques principes de programmation fonctionnelle, à partir du langage `Caml` , afin de rendre plus familière ce type de programmation et de permettre un accès plus aisé aux ouvrages spécialisés sur ce thème. Que les théoriciens m'excusent donc pour les allusions et approximations faites dans ce document, ceci n'est pas une bible, mais simplement un moyen pour les étudiants d'accéder plus facilement aux écrits théoriques et à la programmation fonctionnelle effective.

1.2 Choix du langage support

La programmation fonctionnelle est une « façon » de programmer, et non un langage de programmation particulier. Le standard `ML` introduit en 1977 est une tentative de standardiser ce type d'approche, en se basant sur une théorie de l'informatique appelée le λ-calcul typé. C'est cette théorie qui permet de déterminer ce que peuvent calculer les programmes fonctionnels, mais cela sort du cadre de ce document. Bref, il y a plusieurs façon de faire de la programmation fonctionnelle, on peut même en faire dans presque tous les langages. Certains langages ont une forme dédiée à cette approche, ce sont les langages fonctionnels, dont deux grands noms sont `Lisp` et `ML` . L'avantage de `ML` dans le cadre du contenu de ce document par rapport à `Lisp` est qu'il permet d'introduire proprement la notion de typage, centrale en informatique. Pour des raisons d'esthétisme principalement, nous ne programmerons pas en `ML` , mais en `Caml` , ce qui est la même chose à quelques points de forme près.

1.3 L'eau à la bouche...

Avant de commencer, sachez que la programmation fonctionnelle a pour les informaticiens quelque chose « d'élégant », voire de très génial, et un des objectifs officieux de ce texte est de vous faire toucher du doigt cette grâce. J'espère pour la qualité de votre cohabitation future avec les ordinateurs pouvoir y parvenir.

Chapitre 2

Réécriture

Il y a un principe simple au cœur de la programmation fonctionnelle, c'est la réécriture. Comme pour le vélo, c'est simple quand on sait faire, mais il faut commencer avec des roulettes. C'est ce que vous avez fait durant vos études passées en apprenant les mathématiques. Il y a un rapport étroit entre les calculs faits par un programme fonctionnel et le déroulement d'une démonstration mathématique. On pourra donc dès le début enlever les roulettes, et aborder la programmation fonctionnelle par analogie aux mathématiques.

2.1 Symboles, valeurs et expressions

2.1.1 Valeurs

En mathématiques, on peut calculer l'*expression* (c'est le nom informatique donné aux formules) suivante :

$$\frac{(3+3)\times 5}{5} \tag{2.1}$$

Il suffit de remplacer successivement des « morceaux » par des valeurs. C'est ça que l'on appelle la *réécriture* et qui est le quotidien de tout mathématicien qui fait des calculs. Pour le plaisir, calculons l'expression 2.1. Le symbole $\mapsto$ se lit « se réécrit en ».

$$\frac{(3+3)\times 5}{5} \mapsto \frac{6\times 5}{5} \mapsto \frac{30}{5} \mapsto 6$$

Rien de bien méchant. On peut toutefois poser sur cet exemple simple quelques concepts. Tout d'abord, dans ce calcul, on manipule des *valeurs*, qui sont en fait des éléments particuliers d'un ensemble, les nombres ici. Histoire de ne pas faire comme tout le monde, les informaticiens utilisent le mot *type* pour parler de la notion d'ensemble dans lequel on pioche les valeurs qui sont exprimées dans une expression. Enfin, le calcul consiste comme nous l'avons vu à réécrire un « morceau » en un « morceau plus petit », c'est ce qu'on appelle la *réduction* d'un terme. Cette relation de réduction, qui est l'instrument de la réécriture, vise à transformer des expressions en des valeurs, bref à calculer. On parle donc aussi d'*évaluation* pour nommer les réductions, et on emploiera surtout ce terme.

2.1.2 Symboles

Reste un dernier point, un peu plus épineux, c'est la notion de variable en mathématiques, qui n'apparaît pas dans la formule 2.1. Disons le tout de suite, le concept de variable est délicat en informatique, du fait que ce mot est utilisé dans un tout autre sens qu'en mathématiques. Une variable en informatique, c'est pour résumer une case mémoire avec un nom dessus, ce qui n'a rien à voir (ou peu) avec les variables mathématiques qui existent sans la notion de mémoire. Donc, afin d'éviter des confusions ici, nous appellerons les variables mathématiques *symbole*, et nous n'emploierons plus jamais le mot « variable » une fois cette phrase terminée.

Écrivons donc une formule mathématique avec des varia... heu... des symboles.

$$\frac{(3+x)\times y}{y} \tag{2.2}$$

Une façon de réécrire cette expression, c'est de remplacer tous les symboles par leur valeur, et de se ramener au cas de l'expression 2.1. Cela suppose de disposer d'une correspondance entre des symboles et leur valeurs respectives. Cette correspondance, appelée *liaison*, sera notée $\equiv$. On peut donc évaluer une expression avec des symboles, à condition de disposer d'une série de liaisons symbole/valeur. C'est ce que nous faisons.

$$\begin{aligned}
& \frac{(3+x)\times y}{y} && [x \equiv 3, y \equiv 5] \\
\mapsto\ & \frac{(3+3)\times 5}{5} && [x \equiv 3, y \equiv 5] \\
\mapsto\ & \frac{(6)\times 5}{5} && [x \equiv 3, y \equiv 5] \\
\mapsto\ & \frac{6\times 5}{5} && [x \equiv 3, y \equiv 5] \\
\mapsto\ & \frac{30}{5} && [x \equiv 3, y \equiv 5] \\
\mapsto\ & 6 && [x \equiv 3, y \equiv 5]
\end{aligned}$$

La série de liaisons disponibles pour remplacer les symboles dans une expression, lors de son évaluation, est appelée *environnement*.

Enfin, il est clair qu'au lieu de foncer tête baissée pour remplacer les symboles par leur valeur, il eût été judicieux de s'apercevoir que l'expression se simplifiait par y, ce qui dispense de le remplacer par sa valeur. Ce type de comportement, par opposition aux cas où on évalue frénétiquement les composantes d'une expression, est appelé *évaluation paresseuse*. En programmation fonctionnelle, l'évaluation est souvent non paresseuse, on parle d'*évaluation stricte*, car il est plus simple d'évaluer tout ce qui bouge sans se poser de questions. Néanmoins, il y a des façons de faire malgré tout de l'évaluation paresseuse, qui évalue des expressions que si c'est vraiment nécessaire, c'est-à-dire en dernier recours. Nous y reviendrons.

2.2 Typage

Comme on l'a déjà évoqué, il est tout naturel quand on a un passé en mathématiques, de considérer que les valeurs appartiennent à des ensembles. On a également dit que l'ensemble auquel appartient une valeur sera appelé *type* en informatique. Un des avantages du langage `ML` (et donc de `Caml` utilisé ici) est qu'il repose sur une théorie informatique de la réécriture où la notion de typage est très présente : le λ-calcul typé. Très succinctement, l'idée est que si on connaît les types des constituants d'une expression, on peut *déduire* le type de l'expression. C'est un peu ce que font les physiciens quand ils vérifient l'homogénéité d'une formule. Le type [1] de la quantité de mouvement $p = mv$ est le type de m, des kilogrammes kg, « multiplié par » le type de v. Récursivement, on obtient le type de $v = d/t$ comme le type de d, des mètres m, « divisé par » le type de t, la seconde s. On rassemble les morceaux pour conclure que le type de la quantité de mouvement est $kg \times (m/s)$ soit des $kg.m.s^{-1}$.

Pour trouver le type d'une expression à partir du type de ses constituants, on ne va pas multiplier ou diviser des types, c'est en cela que l'analogie avec les formules physiques est imparfaite, mais on va malgré tout recombiner des types pour déduire des types nouveaux. Attention, nous ferons les recombinaisons à la main pour comprendre, mais ce processus est automatisé en pratique, on a juste à constater le type de l'expression que l'ordinateur a calculé pour nous ! Nous verrons par la suite ce que sont les recombinaisons de types utilisées par `Caml` pour inférer le type d'une expression. C'est ce qu'on appelle l'*inférence de type*.

1. Les physiciens disent unité.

2.3 Utilisation de Caml pour réécrire des expressions

Le logiciel Caml est un interpréteur de commande. Après le caractère #, on est invité à saisir une expression. On termine par ;;, et il retourne une valeur, en indiquant son type. Voici quelques exemples, sachant que l'opérateur ^ représente la mise bout à bout [2] de chaînes de caractères :

```
# 3 + 4;;
- : int = 7
# "toto" ^ "titi";;
- : string = "tototiti"
```

On peut, plutôt que de tout rentrer en ligne depuis le clavier, stocker une suite d'expressions à évaluer dans un fichier. Caml peut ainsi lire ce fichier, et se comporter comme si vous aviez tapé les expressions une à une après le #. Par exemple, si vous tapez dans le fichier schmurtz.ml les lignes suivantes :

```
3 + 4;;
"toto" ^ "titi";;
```

Dans l'interpréteur Caml , vous pouvez lire ce fichier (directive #use) pour que Caml évalue les expressions qui s'y trouvent.

```
# #use "schmurtz.ml";;
- : int = 7
- : string = "tototiti"
```

2.4 Créer des liaisons pour réécrire

Rappelons pour commencer que toute expression est évaluée dans un environnement, c'est-à-dire une série de liaisons symbole/valeur (cf. page 6). Il existe un environnement par défaut, auquel on peut rajouter ses propres liaisons. Nous allons voir quelques moyens de le faire ici, mais il en est d'autres, moins directs, que nous verrons au fur et à mesure de la suite.

2.4.1 Modifier l'environnement définitivement, Soit $x = ...$

La façon la plus simple de créer dans l'environnement courant une liaison symbole/valeur est l'instruction let. Par exemple, les commandes suivantes créent les liaisons $x \equiv 5, y \equiv$ "titi" dans l'environnement.

```
# let x = 2 + 3;;
val x : int = 5
# let y = "titi";;
val y : string = "titi"
```

Caml répond à l'expression let qu'il a effectué une liaison par le mot val, puis il retourne le nom, le type et la valeur impliqués dans cette liaison.

Nota : L'expression à droite de = est évaluée avant de créer la liaison, on retrouve le principe d'évaluation stricte, non paresseuse.

Ayant fait les deux let précédents, observons le comportement de Caml pour l'évaluation d'autres expressions.

```
# x;;
- : int = 5
# y;;
- : string = "titi"
# x+3;;
- : int = 8
```

2. concaténation

```
# let z = y ^ y;;
val z : string = "tititititi"
# let x = x + 3;;
val x : int = 8
# p;;
Unbound value p
# let w = w + 3;;
Unbound value w
```

Notons que si on associe deux fois de suite une valeur à un symbole (c'est le cas du symbole x ici), la première liaison est définitivement perdue. Quand ensuite on écrit `let x = x+3`, l'expression `x+3` est évaluée, avec l'association courante pour le symbole `x`, à savoir $x \equiv 5$. C'est seulement après cette évaluation, qui donne 8, que le symbole `x` est associé à cette nouvelle valeur, on a alors $x \equiv 8$. Donc le fait de définir la valeur de `x` à partir de `x` ne pose pas de problème, du moment que `x` est déjà associé à une valeur dans l'environnement courant. Ce n'est pas le cas de `w` dans l'expression `let w = w+3`, d'où l'erreur.

2.4.2 Modifications temporaires de l'environnement

On peut vouloir créer des liaisons temporaires pour simplifier l'écriture d'une expression.

En math, on écrit parfois $x + \dfrac{e \times f}{e - f}$ où $\begin{cases} e = x + 3 \\ f = 37 \end{cases}$ au lieu de $x + \dfrac{(x+3) \times 37}{x + 3 - 37}$

Les symboles e et f peuvent, voire doivent, n'être définis que le temps d'évaluation de l'expression. On voudrait de plus qu'après l'évaluation, d'éventuelles liaisons antérieures des symboles e et f soient conservées. Pour voir comment faire, construisons d'abord l'environnement suivant :

```
# let e = "sproutch";;
val e : string = "sproutch"
# let f = 'z';;
val f : char = 'z'
# let x = 22;;
val x : int = 22
```

Utilisons l'instruction `let ... and ... in` pour déclarer des liaisons temporaires.

```
# x + let    e = x + 3
         and f =    37
      in
         (e * f) / (e - f);;
- : int = -55
```

Pour bien comprendre, réécrivons l'expression, dans l'environnement
$[e \equiv$ `"sproutch"`$, f \equiv$ `'z'`$, x \equiv 22]$
précédemment construit.

> **Attention !** Ne confondez pas ce que l'on tape en `Caml` avec ce que l'on réécrit ici pour comprendre. Ce que l'on réécrit est fait *implicitement* par `Caml` . Pour bien marquer la différence, nous laisserons au cours de ce document le symbole `#` que `Caml` affiche quand on doit saisir une expression à réécrire, dans les cas où il s'agit de l'expression qu'il faut effectivement taper. Quand nous réécrivons cette expression à la main, pour comprendre ce que `Caml` fait implicitement, *on ne précède pas les expressions* du symbole `#`.

Réécrivons donc l'expression `let ... in` comme le fait `Caml` implicitement.

```
x + let    e = x + 3
       and f =    37
    in
       (e * f) / (e - f);;
```
$[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

```
22 + let    e = 22 + 3
        and f =     37
    in
       (e * f) / (e - f);;
```
$\mapsto$ $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

```
22 + let    e = 25
        and f = 37
    in
       (e * f) / (e - f);;
```
$\mapsto$ $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

Arrêtons nous sur ce point. Nous avons remplacé les x par leur valeur, mais nous ne l'avons pas fait pour les e et les f dans le « corps du `let ... in` ». C'est contraire au principe d'évaluation non paresseuse, où l'on remplace tous les symboles tout de suite ! Eh bien justement, le corps du `let ... in` est une exception, il n'est pas évalué tant qu'on n'en a pas besoin, son évaluation est paresseuse. Or nous nous sommes arrêtés dans notre réécriture manuelle à l'endroit où on a effectivement besoin d'évaluer le corps du `let ... in`. Reprenons donc. L'évaluation de `let ... in` se fait en créant un nouvel environnement que l'on empile sur celui en cours pour évaluer l'expression qui suit le `in`. Pour trouver la valeur d'un symbole, on regarde les environnements « du haut de la pile vers le bas », la valeur du symbole étant fournie par la première liaison trouvée.

```
22 + let    e = 25
        and f = 37
    in
       (e * f) / (e - f);;
```
$[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `22 + (e * f) / (e - f);;` $[e \equiv 25, f \equiv 37]$ $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `22 + (25 * 37) / (25 - 37);;` $[e \equiv 25, f \equiv 37]$ $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `22 + (25 * 37) / (25 - 37);;` $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `22 + 925 / (-12);;` $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `22 + -77;;` $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

$\mapsto$ `-55;;` $[e \equiv \texttt{"sproutch"}, f \equiv \texttt{'z'}, x \equiv 22]$

2.5 Les fonctions

2.5.1 Les valeurs fonctionnelles

Les fonctions en `Caml` ont ceci de commun avec les fonctions mathématiques qu'elles prennent *UN* élément d'un ensemble, pour fournir un résultat qui est un élément d'un ensemble. Ce qu'on appelle une fonction à plusieurs arguments usuellement est en fait une fonction qui prend ses éléments dans un ensemble qui est le produit cartésien de plusieurs autres, c'est-à-dire une fonction qui travaille sur des n-uplets.

Comme tout le reste, une fonction en `Caml` est une valeur... C'est un peu bizarre, mais on peut voir cette valeur comme une description du calcul à effectuer, tout comme une valeur entière est une description d'une quantité. Cette description est une expression, où un symbole joue le rôle particulier de paramètre, ou d'argument. Une valeur fonctionnelle s'écrit `function symbol`

`-> expression`. Illustrons cela sur un cas simple, la fonction qui à `x` associe `x+1`. Le paramètre est donc `x`, l'expression qui représente le calcul de la fonction est `x+1`. La *valeur* représentant cette fonction est donc :

```
function x -> x + 1
```

C'est bien de définir des fonctions, mais encore faut-il pouvoir les utiliser pour faire des calculs. Rien de plus simple, il suffit pour cela d'écrire successivement la valeur fonctionnelle puis l'expression qui lui servira d'argument. On met des parenthèses pour être sur que `Caml` comprend la même chose que nous....

```
# (function x -> x + 1) 3;;
- : int = 4
```

Afin de comprendre, nous allons évaluer l'application d'une valeur fonctionnelle à un argument. Partons de l'exemple suivant.

```
# let x = "toto";;
val x : string = "toto"
# let y = 3;;
val y : int = 3
# (function x -> x + y) (5 + 5);;
- : int = 13
```

La difficulté illustrée par cette exemple est que le corps de fonction utilise un symbole `y`, et un paramètre `x` qui rentre en conflit avec un symbole `x` déjà défini dans l'environnement. Un peu comme avec les `let..in`, tout le problème consiste à écrire le corps de fonction en empilant un environnement correct. Ça se passe en deux temps, et donc deux empilements d'environnements.

Le premier environnement à empiler est l'environnement de définition de la fonction. Il faut avoir à l'idée que dès que `Caml` rencontre la définition d'une valeur fonctionnelle, il lui associe une copie de l'environnement courant. C'est cet environnement de définition qui sera empilé le premier au moment de l'utilisation de la fonction. Le deuxième environnement à empiler contient uniquement la liaison entre le paramètre de la fonction et la valeur de l'argument. Ne reste plus alors qu'à évaluer le corps de fonction en utilisant cette pile d'environnement. On le fait à la main.

$$\texttt{(function x -> x + y) (5 + 5)} \quad [x \equiv \texttt{"toto"}, y \equiv 3]$$

$$\mapsto \begin{array}{c} \texttt{(function x -> x + y)} \\ \updownarrow \\ [x \equiv \texttt{"toto"}, y \equiv 3] \end{array} \quad \texttt{10} \quad [x \equiv \texttt{"toto"}, y \equiv 3]$$

La première étape consiste à associer ($\updownarrow$) un environnement à la valeur fonctionnelle, à savoir une copie de l'environnement courant au moment de sa définition, comme nous l'avons vu. Cette liaison est appelée une *fermeture lexicale*[3]. Durant cette étape, on n'évalue pas le corps de fonction `x+y`. On a à nouveau une exception à la règle d'évaluation stricte ! Le corps de fonction fera l'objet d'une *évaluation paresseuse*, ce qui sera fort utile dans certains cas. Continuons donc, sachant qu'on en arrive maintenant à effectuer le calcul. Il s'agit donc d'empiler l'environnement de définition[4],

3. « closure » en anglais.
4. La fermeture lexicale.

le paramètre formel, puis d'évaluer le corps de fonction.

$$\begin{array}{lll}
\begin{array}{l}\texttt{(function x -> x + y)}\\ \qquad\updownarrow\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} & \texttt{10} & [x \equiv \texttt{"toto"}, y \equiv 3] \\
\mapsto \begin{array}{l}\texttt{(function x -> x + y)}\\ \qquad\updownarrow\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} & \texttt{10} & \begin{array}{l}[x \equiv \texttt{"toto"}, y \equiv 3]\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} \\
\mapsto \begin{array}{l}\texttt{(function x -> x + y)}\\ \qquad\updownarrow\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} & \texttt{10} & \begin{array}{r}[x \equiv 10]\\ [x \equiv \texttt{"toto"}, y \equiv 3]\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} \\
\mapsto \texttt{x + y} & & \begin{array}{r}[x \equiv 10]\\ [x \equiv \texttt{"toto"}, y \equiv 3]\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} \\
\mapsto \texttt{10 + 3} & & [x \equiv \texttt{"toto"}, y \equiv 3] \\
\mapsto \texttt{13} & & [x \equiv \texttt{"toto"}, y \equiv 3]
\end{array}$$

En pratique, on aura tendance à définir une fonction, puis à l'appliquer à plusieurs valeurs. Comme la fonction est une valeur, définir une fonction consiste à associer un symbole à cette valeur, ce qu'on sait faire. Par exemple :

```
# let x = "toto";;
val x : string = "toto"
# let y = 3;;
val y : int = 3
# let f = function x -> x + y;;
val f : on verra le type au paragraphe suivant
# let y = 347;;
val y : int = 347
# f 10;;
- : int = 13
```

La ligne `let f = ...` associe la valeur fonctionnelle à `f`, définissant ainsi une fonction `f`. Mais rappelons nous, cette valeur fonctionnelle est accompagnée de son environnement de définition, où `y` vaut `3`. Au moment d'évaluer `f 10`, `Caml` remplace le symbole `f` par la valeur qu'il a actuellement dans l'environnement, ce qui donne l'expression suivante à réécrire en empilant deux environnements, comme décrit plus haut.

$$\begin{array}{l}\texttt{(function x -> x + y)}\\ \qquad\updownarrow\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array} \quad \texttt{10} \quad \left[\begin{array}{l} x \equiv \texttt{"toto"}, y \equiv 347, f \equiv \begin{array}{l}\texttt{(function x -> x + y)}\\ \qquad\updownarrow\\ [x \equiv \texttt{"toto"}, y \equiv 3]\end{array}\end{array}\right]$$

Comme vous le constatez, c'est un peu lourd à écrire, aussi remplacerons nous les valeurs fonctionnelles qui apparaissent dans les liaisons d'un environnement par `<fun>` : $[x \equiv \texttt{"toto"}, y \equiv 347, f \equiv \texttt{<fun>}]$.

En pratique, ne vous posez pas toutes ces questions. Retenez que l'expression qui sert de corps de fonction (après le `->`) décrit le calcul à faire, et qu'à part le symbole correspondant au paramètre formel, les autres symboles sont complètement verrouillés sur leur valeur au moment de la définition de la fonction.

2.5.2 Le type d'une fonction

Comme toute valeur en `Caml` , les valeurs fonctionnelles `function ... -> ...` ont un type. Une fonction mathématique, d'un ensemble E vers un ensemble F, appartient à l'ensemble des

« fonctions de E vers F ». Traduit en informatique, c'est-à-dire en remplaçant le mot « ensemble » par « type », cela revient à dire qu'une fonction dont l'argument est de type `t1` et qui donne un résultat de type `t2`, est une valeur fonctionnelle de type « `t1` vers `t2` ». `Caml` note ce *type de fonction* `t1 -> t2`. Étudions quelques exemples, sachant que l'opérateur `^` représente la mise bout à bout [5] de chaînes de caractères :

```
# let f = function x -> x + 3;;
val f : int -> int = <fun>
# let g = function x -> x ^ ".ml";;
val g : string -> string = <fun>
# f 10;;
- : int = 13
# g "toto";;
- : string = "toto.ml"
```

Dans le cas de la fonction `f`, on voit que `Caml` trouve son type *tout seul*, c'est le principe d'*inférence de type* que nous avons évoqué en début de paragraphe 2.2. Comment ça marche ? En fait, le type d'une fonction est contraint par l'expression derrière le `->`. Dans le cas de `f`, de l'expression `x + 3`, on peut déduire que comme `+` est un opérateur travaillant sur deux entiers (type `int`), `x` est une expression *nécessairement entière*. Donc la fonction `f` est de type `int -> ...`. Pour ce qui est du résultat, `+` donne un résultat de type `int` également, donc `f` est de type `... -> int`. D'où le type `int -> int` calculé par `Caml` pour `f`. De façon analogue, `Caml` déduit que `g` est une fonction de type `string -> string`.

2.5.3 Les fonctions qui rendent une fonction

On a vu qu'une fonction prend un argument et retourne une valeur. Un cas intéressant est le cas où cette valeur est elle-même une valeur fonctionnelle. Partons de l'exemple suivant :

```
# let plus = function x -> function y -> x + y;;
val plus : int -> int -> int = <fun>
```

On aurait dû écrire `function x -> (function y -> x + y)`, mais on peut se passer des parenthèses. Comme `+` est un opérateur sur deux `int`, `Caml` infère que `x` et `y` sont de type `int`. Donc `function y -> x + y`, *qui est la valeur retournée par la fonction* `plus`, est de type `int -> int`. La fonction `plus` prend donc un argument `x` de type `int`, et rend une valeur de type `int -> int`. Le type de `plus` est donc `int -> (int -> int)`. Quand les parenthèses sont à droite, `Caml` les élimine pour plus de lisibilité, d'où le type `int -> int -> int` retourné. Pour utiliser la fonction `plus`, il faut lui passer un argument. On récupère une fonction, à laquelle on peut également passer un argument, puis on récupère un résultat. Ça donne :

```
# (plus 10) 15;;
- : int = 25
```

Dans ce cas, on peut se passer des parenthèses, et écrire :

```
# plus 10 15;;
- : int = 25
```

On a là une façon détournée de faire une fonction à deux arguments ! Avec ce que l'on sait sur la réécriture des valeurs fonctionnelles (cf. 2.5.1), on peut réécrire l'expression `(plus 10) 15`, où on laisse les parenthèses pour s'assurer de réécrire les choses comme `Caml` le fait. L'environnement avant la définition `let plus = ...` est noté ENV. Après la définition de la fonction, on a l'environnement suivant :

$$\left[\begin{array}{c} \texttt{function x -> function y -> x + y} \\ plus \equiv \quad \updownarrow \qquad\qquad\qquad , \text{ENV} \\ [\text{ENV}] \end{array}\right]$$

5. concaténation

que l'on écrira :

$$[plus \equiv \texttt{<fun>}, \text{ENV}]$$

Évaluons donc l'expression `(plus 10) 15` :

$$\begin{array}{lll}
& \texttt{(plus 10) 15} & [plus \equiv \texttt{<fun>}, \text{ENV}] \\
\mapsto & \left(\begin{array}{c} \texttt{function x -> function y -> x + y} \\ \updownarrow \qquad\qquad\qquad\quad \texttt{10} \\ [\text{ENV}] \end{array} \right) \texttt{15} & [plus \equiv \texttt{<fun>}, \text{ENV}] \\
\mapsto & \texttt{(function y -> x + y) 15} & \begin{array}{r} [x \equiv 10] \\ [\text{ENV}] \\ [plus \equiv \texttt{<fun>}, \text{ENV}] \end{array}
\end{array}$$

Á ce stade, on vient de résoudre l'application de l'argument `10` à la fonction `plus`, cela donne une valeur fonctionnelle `function y -> x + y`, qui vient d'être créée. On lui associe un environnement de définition qui est l'environnement courant, à savoir toute la pile ! Pour simplifier l'écriture, et comme `x` est le seul symbole autre que le paramètre formel dans le corps de fonction `x + y`, on considérera l'environnement de définition comme $[x \equiv 10]$, ce qui ne change rien et est plus simple à écrire. Comme à ce stade on a terminé la réécriture de `plus 10`, on peut enlever les deux environnements empilés, mais pas l'environnement de définition de `function y -> x + y`, car il en garde justement la trace. Cela donne :

$$\left(\begin{array}{c} \texttt{function y -> x + y} \\ \updownarrow \\ [x \equiv 10] \end{array} \right) \texttt{15} \qquad [plus \equiv \texttt{<fun>}, \text{ENV}]$$

On voit ici, et on s'en servira par la suite, que `plus 10` est la fonction « qui attend l'argument `y` pour calculer `10+y` », où autrement dit « qui attend l'argument `y` pour calculer `x+y` où `x` vaut `10` ». Cette remarque faite, terminons la réécriture :

$$\begin{array}{lll}
& \left(\begin{array}{c} \texttt{function y -> x + y} \\ \updownarrow \\ [x \equiv 10] \end{array} \right) \texttt{15} & [plus \equiv \texttt{<fun>}, \text{ENV}] \\
\mapsto & \texttt{x + y} & \begin{array}{r} [y \equiv 15] \\ [x \equiv 10] \\ [plus \equiv \texttt{<fun>}, \text{ENV}] \end{array} \\
\mapsto & \texttt{10 + 15} & [plus \equiv \texttt{<fun>}, \text{ENV}] \\
\mapsto & \texttt{25} & [plus \equiv \texttt{<fun>}, \text{ENV}]
\end{array}$$

Bien sûr en pratique, pas besoin de garder en tête tous ces mécanismes pour concevoir la fonction `plus`. Le raisonnement intuitif est le suivant. Je veux faire une fonction à deux arguments qui les additionne, j'écris `function x -> function y ->` pour prendre les deux arguments, et le corps de fonction est naturellement `x+y`.

Là où tout cela devient intéressant, c'est lorsque on veut définir la fonction `incr` qui ajoute 1. Naïvement, on écrirait :

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
```

Alors qu'il est plus élégant d'écrire :

```
# let incr = plus 1;;
val incr : int -> int = <fun>
```

Dans ce dernier exemple, on définit la fonction `incr` en appliquant un seul des deux arguments attendus par `plus`. C'est ce qu'on appelle l'*application partielle*. Intuitivement, le fait de définir la fonction à deux arguments `plus` avec `function x -> function y -> ...` permet de dire que si on lui donne un seul argument, on obtient la fonction « qui attend le deuxième argument pour rendre le résultat ». Ce mécanisme est rare dans les langages informatiques, alors qu'il est très puissant.

2.5.4 Les fonctions qui prennent une fonction

Nous avons vu au paragraphe précédent que les valeurs fonctionnelles peuvent être des valeurs retournées par une fonction, ce qui crée implicitement des fonctions à deux arguments, avec possibilité d'application partielle. Ici au contraire, on s'intéresse au cas où la fonction prend une valeur fonctionnelle comme argument. On ne va pas tout réécrire, quoique c'est un bon exercice. Contentons nous donc de faire confiance à `Caml` , pour définir une fonction `applique_a_2_et_ajoute_3` qui prend en argument une fonction `f`, l'applique à l'argument 2, et ajoute 3 au résultat. Cela s'écrit simplement en `Caml` :

```
# let applique_a_2_et_ajoute_3 = function f -> (f 2) + 3;;
val applique_a_2_et_ajoute_3 : (int -> int) -> int = <fun>
```

Voyez le type retourné ! `(int -> int) -> int` signifie que `applique_a_2_et_ajoute_3` prend une fonction pour rendre un entier, cette fonction elle-même prenant un entier pour rendre un entier. Les parenthèses sont cruciales ! En effet, le type `int -> int -> int`, qui est en fait `int -> (int -> int)` est celui d'une fonction à deux arguments entiers, comme nous l'avons vu précédemment.

Là où le mécanisme d'inférence de type est fort, c'est que le type de `applique_a_2_et_ajoute_3` est inféré automatiquement. Comment ? Partons du corps de la fonction `(f 2) + 3`. On a le fait que `+` travaille sur deux entier qui impose que `(f 2)`, le résultat rendu par `f`, soit entier. Donc le type de `f` est `...->int`. Pour l'argument de `f`, le fait d'écrire `(f 2)` impose que `f` prenne un argument entier, donc le type de `f` est finalement `int -> int`. Comme `+` rend une valeur entière, `(f 2) + 3` est un résultat de type `int`. Donc l'argument `f` de la fonction `applique_a_2_et_ajoute_3` est de type `int -> int`, son résultat de type `int` d'où le type `(int -> int) -> int` inféré par `Caml` .

Si maintenant on veut définir la fonction `applique_a_2`, qui est la même que précédemment si ce n'est qu'on ne prend pas la peine d'ajouter 3. On écrirait :

```
# let applique_a_2 = function f -> (f 2);;
```

On va trouver comme précédemment en guise de type de `applique_a_2` le type `(int -> int) -> int`... Vérifions quand même avec ce même raisonnement. Le fait d'écrire `(f 2)` m'assure que `f` prends un entier en argument. Son type est donc `int -> ...`. Le type du résultat de `f` est... Tiens ! Alors que dans l'exemple précédent le `+` de l'expression `(f 2) + 3` permettait de conclure que `(f 2)` est un `int`, ici on est coincé. Essayons de poursuivre quand même le raisonnement, en postulant que `(f 2)` est d'un type `A`. La fonction `f` donne suivant cette hypothèse un résultat de type `A`. On tâchera d'associer `A` à un vrai type à la fin du raisonnement, mais on est obligé de dire pour l'instant que `f` est de type `int -> A`. La fonction `applique_a_2` ayant pour corps de fonction `(f 2)`, elle retourne une valeur de type `A`. Donc elle prend un argument `f` de type `int -> A` et retourne une valeur de type `A`, son type est `(int -> A) -> A`. Contrairement à ce qu'on espérait, on arrive en fin de raisonnement sans pouvoir dire quel est ce type `A`. Voyons ce que dit `Caml` .

```
# let applique_a_2 = function f -> (f 2);;
val applique_a_2 : (int -> 'a) -> 'a = <fun>
```

On constate que `Caml` aussi, laisse le type `A` non résolu, le nommant `'a` ! La signification est que `applique_a_2` peut prendre n'importe quelle fonction en argument, du moment que celle-ci prend un entier en argument. Croyant définir une fonction `applique_a_2` qui prend un argument de type `int -> int`, on a en fait définit une fonction `applique_a_2` bien plus générale. C'est souvent le cas lorsqu'on utilise des fonctions comme argument. Nous reviendrons sur les types du genre `'a` dans le paragraphe 3.7. Juste pour se mettre l'eau à la bouche, testons les évaluations suivantes.

```
# let f = function x -> x + 1;;
val f : int -> int = <fun>
# applique_a_2 f;;
- : int = 3
```

Puis essayons la suivante, sachant que la fonction `string_of_int` transforme un entier en une chaîne de caractères (`10` devient `"10"` par exemple) :

```
# let f = function n -> "j'ai " ^ (string_of_int n) ^ " ans.";;
val f : int -> string = <fun>
#  f 30;;
- : string = "j'ai 30 ans."
# applique_a_2 f;;
- : string = "j'ai 2 ans."
```

Á méditer...

Chapitre 3

Les Types

Nous avons vu que la notion de type est centrale en `Caml` , du fait du mécanisme d'inférence de type. Ce mécanisme permet de construire le type d'une expression à partir du type de ces constituants. Nous allons voir dans ce chapitre comment construire des types plus complexes à partir de types simples. Il y a d'ores et déjà une chose à retenir. Ce que `Caml` sait faire pour un type, il saura le faire pour n'importe quel type, aussi compliqué soit-il. C'est ce qu'on appelle la *complétude des types*.

3.1 Définition de type

Nous verrons comment exprimer de nouveaux types à partir de types existants. Montrons auparavant comment on peut leur donner un nom. Le nom de type est aux types ce que le symbole est à la valeur, mais ces deux phénomènes de nommage sont bien distincts. Il y a la même différence en mathématiques entre ces deux nommages :

$$\text{Soit } x = 47$$

$$\text{Soit } \mathcal{A} = \{x \in \mathbb{R} \text{ tq } x > 47\}$$

Pour bien marquer la différence, on n'utilisera pas l'instruction `let` pour donner un nom à un type, mais l'instruction `type`. Voici des exemples de renommage de certains types pré-définis, on verra des cas plus pertinents plus avant.

```
# type reel = float;;
type reel = float
# 3.0;;
- : float = 3
# type chaine = string;;
type chaine = string
# "toto";;
- : string = "toto"
```

Cet exemple illustre juste l'utilisation de `type`, car `Caml` se réfère aux types habituels pour typer les expressions.

3.2 Les n-uplets

La notion de n-uplets en mathématique correspond aux éléments d'un ensemble, qui est un produit cartésien d'autres ensembles. En `Caml` , le triplet $(3, 2, 5)$ s'écrit comme en mathématiques, à savoir `(3,2,5)`. Tout comme l'ensemble auquel $(3, 2, 5)$ appartient, à savoir $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$, le type de `(3,2,5)` en `Caml` est `int * int * int`. Regardons les évaluations suivantes, ainsi que leur type.

```
# 3,4,5;;
- : int * int * int = 3, 4, 5
#  3,"toto",(false,true,('z',3.5));;
- : int * string * (bool * bool * (char * float))
    = 3, "toto", (false, true, ('z', 3.5))
# type point = (int * int);;
type point = int * int
```

3.3 Union

Tout comme nous avons vu le produit cartésien, il existe un moyen de représenter l'union ensembliste de types. Pour le coup, il ne s'agit pas tout à fait de la même notion qu'en mathématique. Voyons le problème. Faisons l'union des deux ensembles $\mathcal{A}$ et $\mathcal{B}$ suivants :

$$\begin{aligned} \mathcal{A} &= \{1, b, \texttt{"toto"}\} \\ \mathcal{B} &= \{\texttt{false}, 1, \texttt{'z'}, b\} \\ \mathcal{A} \cup \mathcal{B} &= \{1, b, \texttt{"toto"}, \texttt{false}, \texttt{'z'}\} \end{aligned}$$

Les deux éléments 1 et b communs aux deux ensembles ne sont pas dupliqués dans l'ensemble union, rien de choquant pour un mathématicien. Les unions ensemblistes que nous ferons en `Caml` sont différentes, dans la mesure cette union est la suivante :

$$\begin{aligned} \mathcal{A} &= \{1, b, \texttt{"toto"}\} \\ \mathcal{B} &= \{\texttt{false}, 1, \texttt{'z'}, b\} \\ \mathcal{A} \cup \mathcal{B} &= \{1, b, \texttt{"toto"}, \texttt{false}, 1, \texttt{'z'}, b\} \end{aligned}$$

Mais en voyant un des éléments dupliqués, on ne sait plus de quel ensemble il provient, et la redondance des éléments en commun crée une ambiguïté peu satisfaisante... La solution consiste à « coller une étiquette » aux éléments des ensembles avant union. Disons que l'on colle l'étiquette ♣ à l'ensemble $\mathcal{A}$, et l'étiquette ♡ à l'ensemble $\mathcal{B}$. On écrit alors :

$$\begin{aligned} \clubsuit\mathcal{A} &= \{\clubsuit 1, \clubsuit b, \clubsuit\texttt{"toto"}\} \\ \heartsuit\mathcal{B} &= \{\heartsuit\texttt{false}, \heartsuit 1, \heartsuit\texttt{'z'}, \heartsuit b\} \\ \clubsuit\mathcal{A} \cup \heartsuit\mathcal{B} &= \{\clubsuit 1, \clubsuit b, \clubsuit\texttt{"toto"}, \heartsuit\texttt{false}, \heartsuit 1, \heartsuit\texttt{'z'}, \heartsuit b\} \end{aligned}$$

Du fait des étiquettes, il n'y a plus d'ambiguïté entre ♣1 et ♡1, ni entre ♣b et ♡b.

On peut avoir besoin de ce type de mécanisme pour regrouper dans un même type des types divers. Par exemple, le type nombre est soit des entiers, soit des réels, soit des complexes, c'est-à-dire des éléments de $\mathbb{R} \times \mathbb{R}$. Il nous faut donc réunir les types `int`, `float`, et `float * float`. On leur « collera » les étiquettes respectives `Entier`, `Reel` et `Complexe`. L'opérateur d'union sur les types est `|`, et l'opérateur d'attribution d'étiquette est `of`. On a alors la définition du type `nombre` suivant :

```
# type nombre =   Entier   of int
                | Reel     of float
                | Complexe of (float * float);;
type nombre = Entier of int | Reel of float | Complexe of (float * float)
# 3;;
- : int = 3
# Entier 3;;
- : nombre = Entier 3
# 3.0;;
- : float = 3
# Reel 3;;
This expression has type int but is here used with type float
# Reel 3.0;;
```

```
- : nombre = Reel 3
# Complexe (3.0,5.18);;
- : nombre = Complexe (3, 5.18)
# ("toto", Complexe (2.5, 6.789));;
- : string * nombre = "toto", Complexe (2.5, 6.789)
```

Là où le mécanisme devient très puissant, c'est qu'on peut utiliser l'ensemble $\mathcal{A}$ lors de la définition de l'ensemble $\mathcal{A}$ lui-même ! C'est déroutant au premier abord, car cela revient à écrire :

$$\mathcal{A} = \heartsuit\mathcal{B} \cup \clubsuit\underline{\underline{\mathcal{A}}} \cup \cdots$$

Cela est utile pour définir des ensembles un peu bizarre. Voyons par exemple une définition du type `humain`, défini soit par le nom de l'individu, donc le type `string` accolé de l'étiquette `Nom`, soit par sa relation à d'autres humains. Cette relation peut être le fait d'être le père de quelqu'un, ce qui se traduit par l'étiquette `Pere` accolée à un objet de type `humain`... type que l'on est en train de définir. De même pour la relation `Mere`. On ajoutera la relation `Enfant` en collant l'étiquette `Enfant` devant un couple d'`humain`s, ses parents, c'est-à-dire un élément de l'ensemble `humain * humain`. Voyons la déclaration du type `humain` :

```
# type humain =   Nom     of string
                | Pere    of humain
                | Mere    of humain
                | Enfant  of humain * humain;;
type humain =
    Nom of string
  | Pere of humain
  | Mere of humain
  | Enfant of humain * humain
```

Essayons de définir quelques personnes par leur nom.

```
# let jean = Nom "Jean";;
val jean : humain = Nom "Jean"
# let pierre = Nom "Pierre";;
val pierre : humain = Nom "Pierre";
# let marie = Nom "Marie";;
val marie : humain = Nom "Marie"
```

Créons maintenant l'humain qui est le père de Jean, ainsi que son grand-père maternel :

```
# let papa_jean = Pere jean;;
val papa_jean : humain = Pere (Nom "Jean")
let pepe = Pere (Mere jean);;
val pepe : humain = Pere (Mere (Nom "Jean"))
```

Pierre et Marie ont eu une fille, qui bien que l'affaire ait été étouffée à l'époque, a eu un enfant illégitime du grand père maternel de Jean, à savoir `pepe`. Nous l'appellerons `monsieur_x` pour ne pas ternir sa réputation.

```
# let fille = Enfant (pierre, marie);;
val fille : humain = Enfant (Nom "Pierre", Nom "Marie")
# let monsieur_x = Enfant (pepe,fille);;
val monsieur_x : humain =
  Enfant (Pere (Mere (Nom "Jean")), Enfant (Nom "Pierre", Nom "Marie"))
```

Vous constatez que `Caml` ne s'embrouille absolument pas dans tous ces ragots.

Le point à retenir dans le cas des types union définis à partir d'eux-mêmes, c'est que pour qu'une valeur existe effectivement, il faut qu'il y ait un des constituants du type qui ne soit pas auto-référent, c'est-à-dire qui ne fasse pas appel au type lui-même. Ici, il s'agit de `Nom of string`.

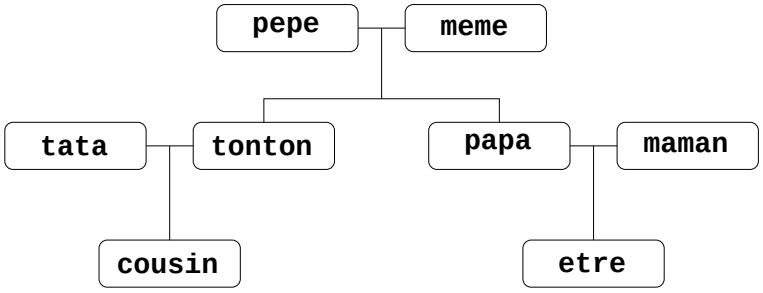

FIGURE 3.1 – Arbre généalogique. Toutes les personnes, exceptée `tata`, s'expriment directement en fonction de `etre`.

Définissons enfin la fonction `cousin_pere`, qui utilisera des personnes nommées `XXX` pour les parents non connus impliqués. Cette fonction prend un `humain` en argument, disons `etre`, et construit son cousin du côté paternel. Voyons cela sur un arbre généalogique (cf. figure 3.1).

Lançons nous dans l'écriture de la fonction :

```
# let cousin_pere = function etre -> ...
```

C'est difficile d'exprimer que le cousin est le fils de l'oncle, sachant que l'oncle est le fils des parents du père (cf. figure 3.1). C'est le moment de se simplifier la vie avec les `let...in` (cf. 2.4.2).

```
# let cousin_pere = function etre ->
       let     pepe = Pere (Pere etre)
           and meme = Mere (Pere etre)
       in
          let     tonton = Enfant (pepe, meme)
              and tata   = Nom "XXX"
          in
              Enfant (tonton,tata);;
val cousin_pere : humain -> humain = <fun>
# cousin_pere (Nom "Marie");;
- : humain = Enfant (Enfant (Pere (Pere (Nom "Marie")),
 Mere (Pere (Nom "Marie"))),Nom "XXX")
```

Même si `Caml` ne le fait pas toujours quand il présente le résultat, il vaudrait mieux écrire ce genre de valeur comme suit :

```
Enfant ( Enfant ( Pere (Pere (Nom "Marie")),
                  Mere (Pere (Nom "Marie"))
                ),
         Nom "XXX"
       )
```

Essayez pour le cousin de `monsieur_x` :

```
# cousin_pere monsieur_x;;
- : humain = Enfant (Enfant (Pere (Pere (Enfant (Pere (Mere (Nom "Jean")),
       Enfant (Nom "Pierre", Nom "Marie")))), Mere (Pere (Enfant (Pere
       (Mere (Nom "Jean")), Enfant (Nom "Pierre", Nom "Marie"))))),Nom "XXX")
```

3.4 Singleton

3.4.1 Définition

Il y a en `Caml` un type qui correspond à un ensemble à un seul élément. Il s'agit d'un élément dont la valeur en elle-même n'a pas d'importance, il suffit juste qu'elle existe. Cette valeur est notée `()`, et le type qui contient cette unique valeur est le type `unit`.

```
# ();;
- : unit = ()
```

Il est clair qu'une fonction qui retournerait une valeur de ce type n'a pas grand intérêt ! En fait cela en a un, mais pas dans le cadre de la programmation fonctionnelle, basée sur la réécriture de valeur. Pour ne pas laisser trop ce point en suspens, disons qu'une fonction qui modifie l'état de quelque chose, n'a pas besoin de retourner une valeur. Il n'empêche pour autant qu'elle est utile, parce que justement elle a modifié un état. Mais ceci est hors sujet pour l'instant.

3.4.2 Étiquettes

Une utilité du type `unit` est qu'il peut participer à la constitution d'un type union, afin de permettre de manipuler des étiquettes « sans valeur accolée [1] ». On définit ainsi ce qu'on appelle des *types énumérés*.

```
# type jour =   Lundi    of unit
              | Mardi    of unit
              | Mercredi of unit
              | Jeudi    of unit
              | Vendredi of unit
              | Samedi   of unit
              | Dimanche of unit;;
type jour =
    Lundi of unit
  | Mardi of unit
  | Mercredi of unit
  | Jeudi of unit
  | Vendredi of unit
  | Samedi of unit
  | Dimanche of unit
# Lundi ();;
- : jour = Lundi ()
```

Dans la définition du type, on peut omettre le `of unit`. Voici un type bien plus utile :

```
# type belote = Sept | Huit | Neuf | Dix | Valet | Dame | Roi | As;;
type belote = Sept | Huit | Neuf | Dix | Valet | Dame | Roi | As
```

Le rôle de ces types dans la construction de types union peut être d'exprimer un cas particulier. Par exemple en mathématique, un entier peut être défini comme 0, ou comme le successeur d'un entier [2]. Cela donne en `Caml` :

```
# type entier =   Zero
                | Succ of entier;;
```

L'entier 3 est le successeur du successeur du successeur de 0.

```
# let trois = Succ (Succ (Succ (Zero)));;
val trois : entier = Succ (Succ (Succ Zero))
```

De même dans l'exemple précédent du type `humain` (cf. 3.3), l'exemple de la fonction `cousin_pere` nous amenait à représenter des humains inconnus, que nous avions codé par l'humain particulier `Nom "XXX"`. Il est plus propre pour traiter ce genre de cas de définir le type `humain` et la fonction `cousin_pere` comme suit :

```
# type humain =   Quidam
                | Nom     of string
                | Pere    of humain
```

1. En fait, on accole l'unique valeur `()` du type `unit`, mais comme cette valeur ne porte pas d'information...
2. C'est sur ce genre de construction de $\mathbb{N}$ que se basent les démonstrations par récurrence

```
                | Mere    of humain
                | Enfant  of humain * humain;;
# let cousin_pere = function etre ->
       let     pepe = Pere (Pere etre)
           and meme = Mere (Pere etre)
       in
          let     tonton = Enfant (pepe, meme)
              and tata   = Quidam
          in
              Enfant (tonton, tata);;
```

3.4.3 Fonctions sans argument

Une fonction prenant une valeur de type `unit` est dite *fonction sans argument*, ce qui est à proprement parler faux, vu que `()`, c'est une valeur, ce n'est donc pas rien. On fait néanmoins l'abus de langage car encore une fois, cette valeur ne véhicule pas d'information, elle a seulement pour fonction d'être là. Les fonctions sans argument représentent des constantes :

```
# let cinq = function () -> 5;;
val cinq : unit -> int = <fun>
#cinq ();;
- : int = 5
```

Pas très utile a priori. Notez qu'il y a à la place du paramètre formel, qui devrait être un symbole, une valeur `()`. C'est un cas particulier de quelque chose que nous verrons plus loin (cf. 3.6). Mais vu que ce symbole n'aurait pu être associé qu'à l'unique valeur du type `unit`, on ne s'en souciera pas ici.

En fait, il y a un intérêt à définir des constantes par le truchement de fonctions sans arguments. Il s'agit du cas où on veut forcer `Caml` a faire de l'évaluation paresseuse. Le corps de fonction, nous l'avons vu, n'est évalué qu'au moment où on lui applique un argument, ce qui fait partie des exceptions à la règle d'évaluation stricte (cf. 2.5.1). Pour retarder l'évaluation d'une expression `expr`, on écrira `let f = function () -> expr`. Pour effectivement déclencher l'évaluation, il suffira d'écrire `f ()`.

3.5 Liste

Une *liste* est une succession de taille finie de valeurs, cette taille pouvant varier d'une liste à l'autre. Il y a un ordre dans les éléments. Ça ne veut pas dire que les éléments sont triés, on peut faire des listes de complexes, alors que $\mathbb{C}$ n'est pas muni d'une relation d'ordre. Il y a un ordre dans les éléments signifie simplement qu'ils ont une place dans la liste, premier, deuxième, ...

Définissons une liste d'entiers avec les outils que nous avons présentés au paragraphe 3.3.

```
# type liste_int =   EtPlusRien
                   | Et    of (int * liste_int);;
type liste_int = EtPlusRien | Et of (int * liste_int)
```

Écrivons la liste des résultats du tiercé, disons 13 et 4 et 7. Commençons par la liste vide, qui sert de « germe ». On peut donc écrire, suivant le type `list_int`, la valeur de type `list_int` suivante :

```
EtPlusRien
```

Introduisons les éléments de la liste des résultats du tiercé en commençant par la fin, donc par le `7`. En regardant le type `list_int`, la seule façon de faire apparaître un entier est de coller l'étiquette `Et` devant un couple constitué d'un entier et d'une liste. Pour l'entier, on prendra bien sûr `7`, et pour la liste requise, on prendra la liste vide `EtPlusRien`, ce qui donne pour l'introduction du dernier élément dans la liste :

```
Et (7, EtPlusRien)
```

Mettons maintenant l'avant dernier élément de la liste des résultats du tiercé, à savoir le 4. On construira à l'aide de l'étiquette `Et` un couple dont le premier élément est l'entier 4, et dont le deuxième est la liste que l'on vient de commencer, à savoir `Et (7, EtPlusRien)`. Cela donne :

```
Et (4, Et (7, EtPlusRien))
```

Pour ajouter enfin le premier élément, 13, on recommence la même opération, ajoutant cet élément en tête de la liste en cours de construction via l'utilisation de l'étiquette `Et`. Cela donne :

```
Et (13, Et (4, Et (7, EtPlusRien)))
```

Voilà comment on représente des listes d'entiers à l'aide du type `list_int`. En Caml , on écrit donc pour représenter la liste du tiercé une valeur qui est le fruit du raisonnement que l'on vient de faire :

```
# Et (13, Et (4, Et (7, EtPlusRien)));;
- : liste_int = Et (13, Et (4, Et (7, EtPlusRien)))
```

Le problème est que cette liste n'est valide que pour des listes d'entiers. En fait, on pourrait remplacer tous les `int` de la définition du type par `A`, pour avoir une liste d'éléments de type `A`. On a déjà vu que Caml gère cette indétermination lors de la définition de la fonction `applique_a_2` (cf. 2.5.4), et nous étudierons ce problème au paragraphe 3.7. Définissons néanmoins un type liste général d'éléments *qui sont tous du même type* `A` :

```
# type 'a liste =   EtPlusRien
                  | Et    of ('a * ('a liste));;
type 'a liste = EtPlusRien | Et of ('a * 'a liste)
```

Reprenons alors les évaluations précédentes, plus quelques autres :

```
# Et (13, Et (4, Et (7, EtPlusRien)));;
- : int liste = Et (13, Et (4, Et (7, EtPlusRien)))
# EtPlusRien;;
- : 'a liste = EtPlusRien
# Et ("Pincemi", Et ("Pincemoi", EtPlusRien));;
- : string liste = Et ("Pincemi", Et ("Pincemoi", EtPlusRien))
```

Le type `list` nous venons de faire « à la main » est équivalent au type `list` de Caml , la différence est que Caml utilise une syntaxe plus agréable. La liste vide est notée `[]` au lieu de `EtPlusRien`, et `Et ( 1, Et (2, ...` est noté

```
1::2::...::[]
```

Enfin, on peut aussi représenter les listes par `[1;2;...]` mais c'est juste une tolérance pour une présentation plus claire, le vrai type sous-jacent est bien

```
1::2::...::[]
```

en conservant l'esprit du type `liste` que nous avions défini. Quelques exemples :

```
1::2::3::[];;
- : int list = [1; 2; 3]
[1; 2; 3];;
- : int list = [1; 2; 3]
[];;
- : 'a list = []
#  [ ["Albert"; "Einstein"] ; ["Pierre"; "et"; "Marie"; "Curie"] ;
     ["Max"; "Plank"] ];;
- : string list list =
   [["Albert"; "Einstein"]; ["Pierre"; "et"; "Marie"; "Curie"];
    ["Max"; "Plank"]]
# [1; "toto"; 3; 4; 5];;
This expression has type string but is here used with type int
```

3.6 Le filtrage

3.6.1 Expression

Nous avons vu plusieurs façons de créer de nouvelles liaisons symbole/valeur. On peut modifier l'environnement courant (cf. `let` 2.4.1), créer des liaisons temporaires (cf. `let ... in` 2.4.2), ou appliquer un argument à une fonction (cf. 2.5.1). Nous en voyons ici une autre, très puissante, qui repose sur la forme que peuvent pendre les valeurs en `Caml` , du fait de la richesse des types que l'on peut construire.

Le principe de ces liaisons est appelé *pattern matching*, où appariement de motif. Le mieux est d'illustrer cela sur des exemples. L'instruction qui réalise l'appariement est `match ... with ...`.

Revenons au type `humain` vu au paragraphe 3.3, dans sa version finale.

```
# type humain =   Quidam
                | Nom     of string
                | Pere    of humain
                | Mere    of humain
                | Enfant  of humain * humain;;
```

On veut définir une réécriture qui transforme un `humain` de la forme `Pere X` en `Mere X`, et réciproquement. De plus, si l'`humain` est de la forme `Enfant (X,Y)`, on veut qu'il soit réécrit en `Enfant (Y,X)`. Enfin, si l'`humain` est de la forme `Nom "xxx"`, on veut qu'il soit réécrit en `Nom "xxx !"`. Soit `etre` un symbole de type `humain` déjà défini, On écrira :

```
# match etre with
      Quidam                    -> Quidam
    | Nom nom                   -> Nom (nom ^ " !")
    | Pere enfant               -> Mere enfant
    | Mere enfant               -> Pere enfant
    | Enfant (parent1, parent2) -> Enfant (parent2, parent1);;
```

Cette expression `match etre with ...` se réécrira en fonction de la forme de la valeur de `etre`. Si cette forme correspond (match en anglais) à `Quidam`, `match etre with ...` se réécrit avec l'expression derrière la flèche `->` correspondante, à savoir la valeur `Quidam`. Si la forme est du type `Nom ...`, on utilise le symbole `nom`[3] pour désigner la chaîne de caractère qui vient après. Dans ce cas, du fait de la présence d'un symbole dans ce qui précède la `->`, l'expression qui suit la `->`, à savoir `Nom (nom ^ " !")`, se réécrit avec une liaison `nom` $\equiv \cdots$ temporaire, ce qui revient à un `let ... in`. Chaque ligne du genre `| ... ->` est appelée *filtre*. Par exemple, réécrivons :

```
    match Nom "Gédéon" with
        Quidam                    -> Quidam
      | Nom nom                   -> Nom (nom ^ " !")
      | Pere enfant               -> Mere enfant
      | Mere enfant               -> Pere enfant
      | Enfant (parent1, parent2) -> Enfant (parent2, parent1);;

    let nom = "Gédéon"
↦   in
        Nom (nom ^ " !")

↦   Nom ("Gédéon" ^ " !")

↦   Nom ("Gédéon !")
```

Essayons un autre exemple :

3. On aurait pu l'appeler autrement, `x`, `schmurtz`, etc...

```
    match Enfant (Nom "Pierre" , Enfant (Quidam, Mere (Nom "Julie")) with
        Quidam                    -> Quidam
      | Nom nom                   -> Nom (nom ^ " !")
      | Pere enfant               -> Mere enfant
      | Mere enfant               -> Pere enfant
      | Enfant (parent1, parent2) -> Enfant (parent2, parent1);;

    let     parent1 = Nom "Pierre"
        and parent2 = Enfant (Quidam, Mere (Nom "Julie")
↦   in
        Enfant (parent2, parent1)

↦   Enfant (Enfant (Quidam, Mere (Nom "Julie"), Nom "Pierre"))
```

Supposons que l'on veuille de plus que `Enfant (Pere X, Y)` se réécrive en `X`. On ajoute le filtre `| Enfant (Pere x, y) -> x` :

```
# match etre with
     Quidam                    -> Quidam
   | Nom nom                   -> Nom (nom ^ " !")
   | Pere enfant               -> Mere enfant
   | Mere enfant               -> Pere enfant
   | Enfant (parent1, parent2) -> Enfant (parent2, parent1)
   | Enfant (Pere x, y)        -> x;;
```

Dans ce cas, `Caml` prévient que le dernier filtre ne sera jamais utilisé. En effet, les filtres sont essayés de haut en bas, et toute valeur qui aurait pu être filtrée par `| Enfant (Pere x, y) ->` sera filtrée avant par `| Enfant (parent1, parent2) ->`. On dit que le filtre `| Enfant (parent1, parent2) ->` est plus général que `| Enfant (Pere x, y) ->` . Il faut donc les mettre dans l'ordre inverse, *du plus particulier au plus général*. Si vous vous trompez, pas de problème, `Caml` vous prévient. De même, si vos filtres ne filtrent pas toutes les valeurs du type, c'est-à-dire si vous oubliez des cas, `Caml` vous prévient aussi. Impossible donc de se tromper sur ces points !

Supposons enfin qu'on veuille réécrire les valeurs de la forme `Pere (Mere X)` en `Quidam`, on écrirait un filtre `| Pere (Mere x) -> Quidam`. Réécrivons alors l'expression :

```
    match Pere (Mere (Nom "Paul")) with
        Quidam                    -> Quidam
      | Nom nom                   -> Nom (nom ^ " !")
      | Pere (Mere enfant)        -> Quidam
      | Pere enfant               -> Mere enfant
      | Mere enfant               -> Pere enfant
      | Enfant (Pere x, y)        -> x
      | Enfant (parent1, parent2) -> Enfant (parent2, parent1);;

    let enfant = Nom "Paul"
↦   in
        Quidam

↦   Quidam
```

On voit que dans ce cas, le symbole temporaire `enfant` du `let ... in` est inutile pour évaluer l'expression qui suit le `in`, à savoir `Quidam`. Ce n'est pas grave en soi, mais `Caml` fournit un symbole pour cela, le symbole `_`, de sorte à rendre les filtres plus « jolis ».

```
# match Pere (Mere (Nom "Paul")) with
    ...
  | Pere (Mere _ )        -> Quidam
    ...  ;;
```

3.6.2 Paramètre d'une fonction

Supposons que je veuille écrire une fonction `f` qui ajoute 2 au troisième élément d'une liste d'entiers, rendant ainsi une nouvelle liste. Le filtrage permet d'écrire facilement cette fonction :

```
# let f = function l ->
    match l with
        []              -> []
      | a::b::fin_liste -> a::(b+2)::fin_liste
      | liste           -> liste;;
val f : int list -> int list = <fun>
# f [1;2;3;4;5;6];;
- : int list = [1; 4; 3; 4; 5; 6]
```

Encore une fois, c'est grâce au `+` de `(b+2)` que `Caml` infère que `b` est de type `int`, donc que tous les éléments de la liste le sont, donc que `f` est de type `int list -> int list`.

Il y a une chose qu'il faut bien comprendre. Quand on écrit un filtre du genre

```
a::b::.....::l
```

le dernier symbole, celui qui termine la série, donc `l` ici, est la *liste* des éléments suivants.

Revenons à notre fonction `f`. Il est très fréquent de traiter des cas selon la valeur du paramètre d'une fonction, donc d'avoir une expression `match .. with` en guise de corps de fonction. C'est pourquoi `Caml` permet de se passer du `match` dans ces cas-là, mais ce n'est qu'une facilité d'écriture, rien de plus.

```
# let f = function    []              -> []
                    | a::b::fin_liste -> a::(b+2)::fin_liste
                    | liste           -> liste;;
val f : int list -> int list = <fun>
```

Supposons que l'on veuille définir une fonction qui enlève le premier élément d'une liste. On écrit :

```
# let decapite = function   []         -> []
                          | _ :: reste -> reste;;
val decapite : 'a list -> 'a list = <fun>
```

On retrouve une fonction générale, avec des types non résolus. Définissons la fonction qui enlève les deux premiers éléments d'une liste, en prenant garde d'écrire les filtres du plus particulier au plus général.

```
# let decapite2 = function   []     -> []
                  | x :: y :: reste -> reste
                  | _ :: reste      -> reste;;
```

Ou mieux,

```
# let decapite2 = function   []     -> []
                  | _ :: _ :: reste -> reste
                  | _ :: reste      -> reste;;
```

Définissons une fonction qui enlève les deux premiers éléments d'une liste, seulement si ceux-ci sont identiques. On écrira :

```
# let decapite2 = function   x :: x :: reste -> reste
                           | liste           -> liste;;
```

`Caml` répond que le symbole `x` « is bound several times in this matching ». Le problème est qu'il ne faut pas oublier que le mécanisme sous-jacent est un

```
let     x = ...
    and x = ...
in
    reste
```

Le problème de faire des filtres où on imposerait, du fait de donner le même nom à un symbole, des relations entre les symboles du filtre, relève du concept d'*unification*, et n'est pas disponible en `Caml` pour des raisons d'impossibilité théorique. Ce mécanisme existe néanmoins dans des langages comme `Prolog` , dits *langages logiques*. C'est toutefois hors de notre propos.

3.7 Types polymorphes

Nous avons vu à plusieurs reprises que `Caml` supporte des types avec des paramètres (`'a`, `'b`,...). C'est ce qu'on appelle le *polymorphisme*. Le mécanisme d'inférence de type, est capable lorsqu'il type une expression, de laisser des variables de type inconnues, où de se rendre compte que deux variables de type doivent être identiques. Écrivons pour s'en convaincre une fonction `paire` qui forme un couple, et une fonction `liste` qui forme une liste à deux éléments.

```
# let paire = function x -> function y -> (x,y);;
val paire : 'a -> 'b -> 'a * 'b = <fun>
# let liste = function x -> function y -> [x ; y];;
val liste : 'a -> 'a -> 'a list = <fun>
```

Le type de la fonction `liste` montre que les deux arguments, du fait qu'il vont être regroupés dans une liste, *doivent* être du même type, `'a`. Il n'y a pas cette contrainte pour la fonction `paire`.

Pour définir soi-même un type polymorphe, il suffit de définir un type en précisant le [4] paramètre. Définissons un type polymorphe `couple`, qui soit un couple d'éléments du même type. Ce type polymorphe est au produit cartésien ce que l'élévation au carré est à la multiplication... Cela donne :

```
# type 'a couple = 'a * 'a;;
type 'a couple = 'a * 'a
# 3,5;;
- : int * int = 3, 5
```

On aurait bien aimé avoir comme type de résultat `int couple`. Mais comme `Caml` avait le choix, il a choisi le type `int * int`, qui type tout aussi bien l'expression `3,5`. Une solution dans ces cas-là est de coller une étiquette, juste pour forcer `Caml` à utiliser notre type.

```
# type 'a couple = Couple of 'a * 'a;;
type 'a couple = Couple of 'a * 'a
# Couple (3, 5);;
- : int couple = Couple (3, 5)
# Couple ("toto","titi");;
- : string couple = Couple ("toto", "titi")
#  Couple ("toto", 3);;
This expression (3) has type int but is here used with type string
```

Les `list` sont des types polymorphes, il y en a bien d'autres.

On précisera enfin que dire de `list` que c'est un type est un abus de langage, il s'agit d'un type polymorphe. On peut dire en revanche que `int list` est un type. Les types polymorphes sont par rapport aux type un niveau au dessus, on dit *méta* en logique.

4. ou les, mais en s'en tiendra à un paramètre.

Chapitre 4

Récursivité

La récursivité est une notion centrale en informatique, souvent présentée comme difficile car elle requiert un mode de raisonnement particulier. Nous verrons dans un premier temps comment réécrire des fonctions récursives. Cela permettra de démystifier le côté « magique » de ces fonctions. Dans un deuxième temps seulement, on verra comment concevoir soi-même une fonction récursive pour résoudre élégamment, avec concision, des problèmes qui sans cela pourraient sembler difficiles.

4.1 Fonction récursive et sa réécriture

Au paragraphe 2.5.1, nous avions vu qu'au moment de la définition d'une valeur fonctionnelle, tous les symboles intervenant dans l'expression qui est le corps de fonction sont verrouillés à leur valeur dans l'environnement courant, excepté bien sûr le symbole servant de paramètre formel. Observons l'exemple suivant :

```
# let f = function x -> x + 1;;
val f : int -> int = <fun>
# let f = function x -> (f x) + 1;;
val f : int -> int = <fun>
# f 10;;
- : int = 12
```

Que s'est-il passé ? On définit d'abord `f` comme la fonction qui ajoute 1. On redéfinit ensuite le symbole `f` comme la fonction qui applique `f`, et ajoute 1 au résultat. Du fait du principe de fermeture 2.5.1, le `f` qui apparaît dans `(f x) +1` correspond à la valeur de `f` dans l'environnement courant, à savoir la fonction qui ajoute `1`, celle que nous avons définie en premier. Donc quand on définit pour la deuxième fois `let f = ...`, le `f` dont on parle « derrière la `->` » est « le `f` d'avant », non celui que l'on est en train de définir. Après de deuxième `let`, l'environnement est le suivant :

$$\left[\begin{array}{l} \texttt{function x -> (f x) + 1} \\ \quad \updownarrow \\ \texttt{f} \equiv \left[\begin{array}{l} \texttt{function x -> x + 1} \\ \texttt{f} \equiv \quad \updownarrow \quad\quad , \text{ENV} \\ \quad\quad [\text{ENV}] \end{array}\right] , \text{ENV} \end{array}\right]$$

Essayons maintenant d'aborder la star de la fonction récursive, à savoir la factorielle. On a deux cas : soit l'argument est 0, auquel cas le résultat est 1, soit l'argument est un entier [1] `n`, auquel cas le résultat est `n * fact (n-1)`. On traitera ces deux cas grâce au filtrage de type sur les arguments (cf. 3.6.2).

```
# let fact = function   0 -> 1
                      | n -> n * (fact (n-1));;
Unbound value fact
```

1. Positif, mais nous ne le vérifierons pas ici.

Caml , au moment de construire la fermeture de la fonction `fact`, à savoir l'expression `n * (fact (n-1))` [2], ne trouve pas dans l'environnement courant « d'ancienne valeur de `fact` », contrairement au cas de la deuxième définition de la fonction `f` précédente. Il faut alors lui préciser que le `fact` dont on parle est la valeur fonctionnelle elle-même, que ce n'est pas la peine de chercher un symbole `fact` dans l'environnement courant pour définir la fermeture. Pour représenter cela, on notera le fait qu'un symbole se réfère à la valeur fonctionnelle en cours de définition par SELF. Pour définir des fonctions en profitant de ce mécanisme, on adjoindra au mot `let` la mention `rec`, pour « récursif ».

```
# let rec fact = function   0 -> 1
                          | n -> n * (fact (n-1));;
val fact : int -> int = <fun>
```

On a alors l'environnement suivant :

$$\left[\begin{array}{l} \texttt{fact} \equiv \begin{array}{c} \texttt{function 0 -> 1 | n -> n * (fact (n-1))} \\ \updownarrow \\ [\texttt{fact} \equiv \text{SELF}] \end{array} \end{array}, \text{ENV}\right]$$

Réécrivons alors `fact 4`. Nous ne ferons pas la gestion des environnements, qui est lourde alors qu'une réécriture intuitive est plus claire.

```
     fact 4
 ↦   4 * (fact (4-1))
 ↦   4 * (fact 3)
 ↦   4 * (3 * (fact 2))
 ↦   4 * (3 * (2 * (fact 1)))
 ↦   4 * (3 * (2 * (1 * (fact 0))))
 ↦   4 * (3 * (2 * (1 * 1)))
 ↦   4 * (3 * (2 * 1))
 ↦   4 * (3 * 2)
 ↦   4 * 6
 ↦   24
```

Essayons des fonctions plus originales. On avait défini page 20 un type entier à l'aide du type suivant :

```
# type entier =   Zero
                | Succ of entier;;
type entier = Zero | Succ of entier
```

On considère alors 3 comme `Succ(Succ(Succ(Zero)))`. Construisons une fonction récursive qui associe à un `int` de Caml l'`entier` correspondant.

```
# let rec entier_of_int = function   0 -> Zero
                                   | n -> Succ (entier_of_int (n-1));;
val entier_of_int : int -> entier = <fun>
```

Constatons [3] que ça marche en réécrivant `entier_of_int 4`.

```
     entier_of_int 4
 ↦   Succ (entier_of_int (4-1))
 ↦   Succ (entier_of_int 3)
 ↦   Succ (Succ (entier_of_int 2))
 ↦   Succ (Succ (Succ (entier_of_int 1)))
 ↦   Succ (Succ (Succ (Succ (entier_of_int 0))))
 ↦   Succ (Succ (Succ (Succ Zero)))
```

On peut faire la fonction inverse :

2. Il construit aussi une fermeture pour le cas 0, mais cela ne pose pas de problème ici.
3. On fera plus que simplement constater au paragraphe 4.4.1.

```
# let rec int_of_entier = function   Zero   -> 0
                                   | Succ x -> 1 + (int_of_entier x);;
val int_of_entier : entier -> int = <fun>
```

Constatons que ça marche :

```
     int_of_entier Succ(Succ(Succ(Succ Zero)))
 ↦   1+(int_of_entier Succ(Succ(Succ Zero)))
 ↦   1+(1+int_of_entier (Succ(Succ Zero)))
 ↦   1+(1+(1+int_of_entier (Succ Zero)))
 ↦   1+(1+(1+(1+int_of_entier Zero)))
 ↦   1+(1+(1+(1+0)))
 ↦   1+(1+(1+1))
 ↦   1+(1+2)
 ↦   1+3
 ↦   4
```

Là où ça commence à devenir magique, c'est lorsque on écrit l'addition comme suit, en utilisant un couple pour représenter les deux arguments :

```
#let rec plus = function   (Zero,   n) -> n
                         | (Succ x, n) -> Succ (plus (x,n));;
val plus : entier * entier -> entier = <fun>
```

Avant de constater que ça marche, essayons de refaire l'inférence du type de la fonction. Le type de l'argument de la fonction est donné aussi bien par `(Zero,n)->`, que par `(Succ x,n)->`, qui étant deux configurations possibles de l'argument, doivent être du même type. N'ayant aucune information sur `n` dans les deux cas, nous dirons que l'argument est de type `entier * A`. Regardons maintenant le résultat de la fonction, qui doit être bien entendu du même type dans les deux cas du filtre. Le premier cas retourne `n`, donc la fonction retourne une valeur de type `A`. On peut dire que `plus` est du type `(entier * A) -> A`. Regardons la valeur retournée par le deuxième filtre. Il s'agit de `Succ` de quelque chose, cette chose étant le résultat de `plus`, donc de type `A`. Comme il ne peut y avoir qu'une valeur de type `entier` après `Succ`, on peut déduire que ce type `A` est forcément le type `entier`. On retrouve aussi ce résultat à partir du deuxième filtre, en voyant que la fonction `plus` retourne une valeur `Succ ...`, donc de type `entier`. D'où le type `(entier * entier) -> entier` de `plus`. Constatons, cet aparté sur l'inférence de type étant terminé, que `plus` fait ce qu'on attend en ajoutant 4 et 5.

```
# let quatre = entier_of_int 4;;
val quatre : entier = Succ (Succ (Succ (Succ Zero)))
# let cinq = entier_of_int 5;;
val cinq : entier = Succ (Succ (Succ (Succ (Succ Zero))))
# let neuf = plus (quatre, cinq);;
val neuf : entier =
  Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero))))))))
# int_of_entier neuf;;
- : int = 9
```

Réécrivons ce calcul :

```
     plus ( quatre, cinq )
 ↦   plus ( Succ(Succ(Succ(Succ Zero))) , Succ(Succ(Succ(Succ(Succ Zero)))) )
 ↦   Succ (plus ( Succ(Succ(Succ Zero)) , Succ(Succ(Succ(Succ(Succ Zero)))) ) )
 ↦   Succ (Succ (plus ( Succ(Succ Zero) , Succ(Succ(Succ(Succ(Succ Zero)))) ) ))
 ↦   Succ (Succ (Succ (plus ( Succ Zero , Succ(Succ(Succ(Succ(Succ Zero)))) ) )))
 ↦   Succ (Succ (Succ (Succ (plus ( Zero , Succ(Succ(Succ(Succ(Succ Zero)))) ) ))))
 ↦   Succ (Succ (Succ (Succ ( Succ(Succ(Succ(Succ(Succ Zero)))) ))))
```

A titre d'exercice pénible et laborieux, vérifiez la multiplication par une réécriture à la main :

```
#let rec fois = function   (Zero,   n) -> Zero
                         | (Succ x, n) -> plus (n, fois(x,n));;
val fois : entier * entier -> entier = <fun>
# let six = entier_of_int 6;;
val six : entier = Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
# let huit = entier_of_int 8;;
val huit : entier =
  Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))
# let cinquante_six = fois (six,huit);;
val cinquante_six : entier = ....
# int_of_entier  cinquante_six;;
- : int = 48
```

4.2 Pour ou contre la récursion terminale

Il y a une chose frappante dans la réécriture de `fact` que nous avons faite page 28, chose qui se retrouve souvent quand on réécrit des fonctions récursives, c'est la croissance de l'expression récursive jusqu'au cas non récursif (`fact 0` ici), puis la décroissance de l'expression, durant laquelle le calcul est effectivement fait. Nous ne rentrons pas ici dans des considérations sur la façon dont le logiciel `Caml` est lui-même programmé, mais disons simplement que cette réécriture coûte cher en temps de calcul et en mémoire. Donc notre façon d'écrire `fact` sera lente pour `fact 100`, et occupera la mémoire vive.

C'est un phénomène indésirable, que l'on peut compenser en écrivant la fonction récursive `fact` autrement. On appellera cette nouvelle façon de faire des fonctions récursive *récursion terminale* ou façon *itérative*.

```
# let rec fact_iter = function   (0,res) -> res
                               | (n,res) -> fact_iter (n-1,n*res);;
```

Calculons 5! en utilisant `fact_iter`. Il faut savoir pour cela que le premier élément de la paire passée en argument est le nombre dont on cherche la factorielle, et le deuxième est toujours 1.

```
     fact_iter (5,1)
 ↦   fact_iter (5-1,5*1)
 ↦   fact_iter (4,5)
 ↦   fact_iter (4-1,4*5)
 ↦   fact_iter (3,20)
 ↦   fact_iter (3-1,3*20)
 ↦   fact_iter (2,60)
 ↦   fact_iter (2-1,2*60)
 ↦   fact_iter (1,120)
 ↦   fact_iter (1-1,1*120)
 ↦   fact_iter (0,120)
 ↦   120
```

Le terme réécrit n'explose plus du tout, et on fait le calcul en une passe, c'est-à-dire sans faire une phase d'expansion du terme à réécrire puis une phase de calcul. Il y a malgré tout un *inconvénient majeur* à cette méthode, c'est qu'à voir la définition de la fonction `fact_iter`, on ne sait pas trop ce que ça calcule, alors que la définition de `fact` est limpide. De ce fait, l'utilisation de fonctions itératives, quoique plus efficace, est source de nombreux bugs.

Le dilemme est là. Qu'est-ce qui coûte cher ? Est-ce le temps de développement où le temps d'exécution ? Un programme basé sur des fonctions itératives sera efficace, mais après une longue série de tests et un debugging laborieux, et jamais fiable. Un programme récursif sera un peu moins efficace [4], mais il se prêtera volontiers à la preuve automatique de programme, et nécessite

4. Sauf si le langage possède un mécanisme interne de transformation automatique d'une fonction récursive en une fonction itérative, mais la discussion ici suppose que non.

un développement moins long, car à peu de chose près, surtout grâce au typage fort comme celui fourni par `Caml` , ça marche du premier coup.

Donc pour un utilisateur qui n'en est pas à une demi seconde près pour retrouver une page Web à partir de mots-clé, autant développer plus rapidement à l'aide de fonctions récursives, et avoir ainsi un code plus lisible et plus maintenable. En revanche, pour le développement de bibliothèques de calcul scientifique intensif sur machines parallèles, on préférera la programmation itérative.

4.3 Récursivité croisée

On est parfois amené à définir des fonctions récursivement croisées. Cela signifie que `f` est utilisée dans la définition de `g` et que `g` est utilisée dans la définition de `f`. La solution consiste à définir les deux (ou plus) fonctions d'un coup, dans un même `let rec`. Nous montrons un exemple sans nous attarder sur ce point.

```
# let rec      f = function x -> ... g ...
          and  g = function x -> ... f ... ;;
```

4.4 Méthodologie et exemples

On a vu jusqu'ici comment réécrire des fonctions récursives jusqu'à obtenir un résultat, mais le plus intéressant est d'être capable de concevoir soi-même une fonction récursive. C'est l'objet du paragraphe à venir et des exemples qui lui succèdent. On s'attachera ici à la conception de fonction non itératives, car ce sont les plus élégantes.

4.4.1 Comment concevoir une fonction récursive

L'application d'une fonction à son argument peut être vu comme un service que l'on délègue à `Caml` . Par exemple, quand on écrit `2 + (fact 5)`, on considère qu'on dispose du service `fact`, sans se préoccuper de la façon dont fact est calculé. C'est un peu comme une machine à laver, une fois qu'on appuie sur le bouton, elle lave le linge, peut importe comment, l'essentiel étant que ça sente bon le propre en fin de cycle.

Cette vue des fonctions aide à définir une fonction récursive `f`. La question à se poser est la suivante : J'ai un service `f` à rendre, à partir d'un argument `x` donné. Est-ce que le fait de déjà savoir rendre le service `f` peut m'aider ? Dit comme cela, la réponse est bien sur « oui », et ça n'avance pas à grand chose. Ça paraît évident, mais on a vite fait au début de programmer ce genre de raisonnement « qui n'avance à rien » sans s'en rendre compte. Formulons donc mieux la question, pour qu'effectivement elle *avance* à quelque chose :
J'ai un service `f` à rendre, à partir d'un argument `x` donné. Est-ce que le fait de déjà savoir rendre le service `f` *pour un argument plus petit* peut m'aider à rendre ce service pour `x` ?

Le terme « plus petit » est ce qui fait que ce raisonnement, qui peut sembler tourner en rond, progresse en fait droit vers la solution. Cette progression est inexorable, aussi faut-il ne pas oublier de la stopper sur « le plus petits des arguments possibles ». On appelle cette action de stopper la *condition d'arrêt*, condition qui, quand on l'oublie, conduit à des fonctions récursives qui ne s'arrêtent jamais.

Reprenons la conception de `fact` dans cet esprit. L'argument est `n`, un entier. La question est de savoir si être capable de calculer `fact` pour un entier plus petit que `n` permet de calculer `fact` pour `n`. En général, « plus petit » se comprend comme « juste un cran plus petit », c'est-à-dire `n-1` ici. Donc ma question revient à la suivante : Puis-je calculer `fact n` en disposant du résultat de `fact (n-1)`. Bien entendu, il suffit de multiplier `fact (n-1)` par `n`. C'est simple non ? C'est toujours aussi simple la récursivité, il suffit juste de ne pas avoir peur d'affirmer qu'on sait résoudre le problème pour l'argument « plus petit ». N'oublions pas la condition d'arrêt, à savoir le plus petit entier dont on puisse calculer la factorielle, c'est-à-dire `0`. Pour lui, pas de problème, le résultat vaut `1`. On retrouve donc :

```
# let rec fact = function   0 -> 1
                          | n -> n * (fact (n-1));;
```

```
val fact : int -> int = <fun>
```

Ça paraît évident sur l'exemple de la factorielle, mais ça l'est moins sur l'exemple suivant, qui est une autre grande star du monde des fonctions récursives.

Il s'agit de résoudre le problème des Tours de Hanoï, récréation mathématique publié en 1883 par le mathématicien François-Édouard-Anatole Lucas. Le problème est le suivant : n plateaux circulaires percés en leur centre sont enfilés sur un pieu, du plus gros en bas au plus petit en haut. On dispose de deux autres pieux, vides en début de partie (cf. figure 4.1a). Il s'agit d'amener ces n plateaux du pieux initial vers un des deux autres pieux, en respectant la règle suivante : les plateaux sur un pieux sont toujours rangés des plus gros en bas vers les plus petits en haut. Un mouvement consiste à déplacer un plateau qui est au sommet d'un tas pour l'enfiler sur un autre pieu, sans violer la règle.

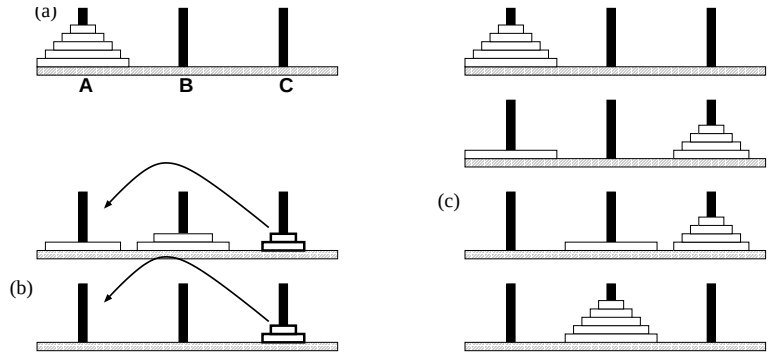

FIGURE 4.1 – Problème des tours de Hanoï.

Résolvons ce problème à l'aide d'une approche récursive. Il faut d'abord se demander quels sont les arguments de la fonction, et quel est son résultat. Le problème est de déplacer n plateaux d'un pieu `depart` vers un pieu `destination`, en utilisant un pieu `pivot`. Le résultat sera une chaîne de caractères, du genre `"... A->B C->B ..."` décrivant les mouvements à faire : ... déplacer le plateau au sommet du pieux A vers le pieux B, puis déplacer le plateau au sommet du pieux C vers le pieux B puis ...

On peut, pour faire beau, définir quelques fonctions et types élémentaires :

```
# type pieu = PieuA | PieuB | PieuC;;
type pieu = PieuA | PieuB | PieuC
# let ecris_mvt = function   (PieuA,PieuB) -> "A->B "
                           | (PieuA,PieuC) -> "A->C "
                           | (PieuB,PieuA) -> "B->A "
                           | (PieuB,PieuC) -> "B->C "
                           | (PieuC,PieuA) -> "C->A "
                           | (PieuC,PieuB) -> "C->B "
                           | (_,_)         -> "error ";;
val ecris_mvt : pieu * pieu -> string = <fun>
# (ecris_mvt (PieuA,PieuC))^(ecris_mvt (PieuC,PieuA));;
- : string = "A->C C->A "
```

La fonction `hanoi` va être définie, selon la façon dont nous avons posé le problème, comme :

```
#let rec hanoi = function (n,depart,destination,pivot) -> ....
val hanoi : int * pieu * pieu * pieu -> string = <fun>
```

Posons nous alors la question miracle, sachant que pour nous, un problème plus petit est un problème avec un plateau de moins. Donc, si on sait déplacer `n-1` plateaux d'un pieu vers un autre, en utilisant un troisième comme pivot, sait-on déplacer `n` plateaux d'un pieu vers un autre, en en utilisant un troisième comme pivot ?

C'est là qu'intervient l'intelligence, en remarquant, comme illustré en figure 4.1b, que le problème de déplacer un ensemble de petits plateaux, en trait gras sur la figure, en respectant la règle du jeu est indépendant des plateaux plus gros, car les petits seront de toute façon toujours au dessus des plus gros.

L'idée est alors la suivante (cf. figure 4.1c). Pour déplacer `n` plateaux de `depart` vers `destination` en utilisant `pivot`, il faut déplacer `n-1` plateaux de `depart` vers `pivot`, ce qu'on sait faire car c'est un problème plus petit, puis faire le mouvement `depart` vers `destination`, puis déplacer à nouveau `n-1` plateaux de `pivot` vers `destination`, en utilisant `depart`. La condition d'arrêt est la suivante : quand on demande à déplacer 0 plateaux, il ne faut rien faire.

Il n'y a plus qu'à l'écrire, en rappelant que `^` est l'opérateur de concaténation de chaîne de caractères :

```
#let rec hanoi = function
     (0,_,_,_)                    -> ""
   | (n,depart,destination,pivot) ->
          (hanoi (n-1,depart,pivot,destination))
         ^(ecris_mvt (depart,destination))
         ^(hanoi (n-1,pivot,destination,depart));;
val hanoi : int * pieu * pieu * pieu -> string = <fun>
```

On essaie :

```
# hanoi (3,PieuA,PieuB,PieuC);;
- : string = "A->B A->C B->C A->B C->A C->B A->B "
# hanoi (5,PieuA,PieuB,PieuC);;
- : string =
"A->B A->C B->C A->B C->A C->B A->B
 A->C B->C B->A C->A B->C A->B A->C
 B->C A->B C->A C->B A->B C->A B->C
 B->A C->A C->B A->B A->C B->C A->B
 C->A C->B A->B "
```

Époustouflant non ?

Enfin, pour clore ce paragraphe de méthodologie, soulignons que la récursivité est utilisée massivement quand les types sont eux-mêmes récursifs, ce qu'on a vu au paragraphe précédent pour les `entier`. Les listes sont des types récursifs par excellence, car une liste est un élément (la tete), suivi de la liste des suivants. Cette liste des suivant joue le rôle du « plus petit » nécessaire à faire avancer la récursion.

L'énorme majorité des fonctions récursives travaillant sur des liste sont des fonctions de la forme :

```
# let rec f = function   []                -> ...
                       | tete :: liste_suiv -> ... ;;
```

On déduit ainsi trivialement la fonction qui compte les éléments d'une liste :

```
# let rec nb_elem = function   []         -> 0
                             | tete::suiv -> 1 + (nb_elem suiv);;
val nb_elem : 'a list -> int = <fun>
# nb_elem ['a'; 'c'; 'a'; 'z'; 'k'; 'h'];;
- : int = 6
```

On appréciera le type polymorphe `'a list -> int`.

4.4.2 Exemples

Coller deux listes bout à bout

On souhaite définir la fonction `bout_a_bout` qui prend deux liste `[A;B;C]` et `[D;E;F]` et retourne la liste `[A; B; C; D; E; F]`. En `Caml` , il suffit d'écrire `[A;B;C]@[D;E;F]`, mais nous allons refaire cette fonction.

Le principe de récurrence est le suivant. Soit

```
tete::suiv
```

la première liste, on suppose qu'on sait mettre bout à bout `suiv` et la deuxième liste. Il suffit de mettre `tete` en tête de ce qu'on vient de calculer, et le tour est joué.

La récurrence porte sur la première des deux listes. Il serait lourd de filtrer cet argument tout de suite, car dans chaque cas, il faudrait reprendre le deuxième argument. Ça donnerait :

```
# let rec bout_a_bout = function   []         -> function l2 -> ....
                                 | tete::suiv -> function l2 -> ....;;
```

Autant « récupérer » les deux arguments d'abord, et filtrer après, avec un `match ... with`.

```
# let rec bout_a_bout = function l1 -> function l2 ->
    match l1 with
        []         -> l2
      | tete::suiv -> tete::(bout_a_bout suiv l2);;
val bout_a_bout : 'a list -> 'a list -> 'a list = <fun>
bout_a_bout ['a'; 'b'; 'c'; 'd'; 'e'] ['f'; 'g'; 'h'];;
- : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h']
```

On notera que `Caml` infère que les deux listes passées en argument doivent contenir des éléments du même type.

Appliquer une fonction a une liste

On veut écrire la fonction `map` qui prend une fonction `f`, une liste, disons `[a; b; c]`, et qui rend la liste `[f a; f b; f c]`, c'est-à-dire la liste des valeurs retournées par l'application de `f` à chacun des éléments de la liste de départ. Pour cela, il suffit d'appliquer `f` au premier élément, et d'ajouter cet élément en tête de la liste résultant de l'application de `map` aux éléments suivants.

```
# let rec map =
    function f ->
    function   []         -> []
             | tete::suiv -> (f tete) :: (map f suiv);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Bien joué pour le type polymorphe inféré !

```
# map string_of_int [1;2;3;4;5];;
- : string list = ["1"; "2"; "3"; "4"; "5"]
# map (function x -> [x;x]) ["1"; "2"; "3"; "4"; "5"];;
- : string list list =
[["1"; "1"]; ["2"; "2"]; ["3"; "3"]; ["4"; "4"]; ["5"; "5"]]
```

Appliquer un opérateur aux éléments d'une liste

Pour concevoir cette fonction, on va raisonner comme si `op` était l'addition, et `neutre` son élément neutre 0. Construisons la fonction `iter` qui somme les éléments d'une liste. Il suffit d'ajouter la tête de liste au résultat de `iter` appliqué aux suivants. Cela donne :

```
# let rec iter =
    function op ->
    function neutre ->
    function   []         -> neutre
             | tete::suiv -> op tete (iter op neutre suiv);;
val iter : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Alors que l'on a raisonné sur addition uniquement, l'inférence de type nous informe qu'il suffit en fait d'avoir un opérateur binaire dont le type du deuxième argument soit identique au type du résultat (`'a -> 'b -> 'b`), ce type étant aussi celui de l'élément neutre. La liste à fournir doit être du type du premier argument attendu par l'opérateur, et le résultat est du type du deuxième. Essayons pour la somme :

```
# iter (function x -> function y -> x + y) 0 [1;2;3;4;5];;
- : int = 15
```

On peut à partir de `iter` définir la fonction `somme`, qui somme les éléments d'une liste. Il suffit d'utiliser l'application partielle d'`iter` (cf. 2.5.3, page 14).

```
# let somme = iter (function x -> function y -> x + y) 0;;
val somme : int list -> int = <fun>
# somme [1;2;3;4;5];;
- : int = 15
```

Bravo encore l'inférence du type de `somme` à partir de celui d'`iter`.

Utilisons `iter` dans un autre cadre que la sommation d'entiers qui nous a guidé dans sa définition.

```
# let colle_entiers_dans_chaine =
    iter (function x -> function y -> (string_of_int x)^y ) "";;
val colle_entiers_dans_chaine : int list -> string = <fun>
# colle_entiers_dans_chaine [1; 2; 78; 79; 31415];;
- : string = "12787931415"
```

Renversement de liste

Pour inverser l'ordre des éléments d'une liste, il suffit de mettre le premier élément à la queue de l'inversée des suivants. Donc :

```
# let rec a_la_queue =
    function elem ->
    function   []         -> [elem]
             | tete::suiv -> tete :: (a_la_queue elem suiv);;
val a_la_queue : 'a -> 'a list -> 'a list = <fun>
# let rec renverse =
    function   []         -> []
             | tete::suiv -> a_la_queue tete (renverse suiv);;
val renverse : 'a list -> 'a list = <fun>
# renverse [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]
```

Cette façon de renverser une liste est très inefficace, du fait d'un appel à chaque étape de la récursion de renverse à une fonction récursive `a_la_queue`. C'est dans ce genre de cas qu'on peut risquer une version itérative.

```
# let rec renverse_it =
    function   ([],resultat)         -> resultat
             | (tete::suiv,resultat) -> renverse_it (suiv, tete::resultat);;
val renverse_it : 'a list * 'a list -> 'a list = <fun>
# renverse_it ([1; 2; 3; 4; 5], []);;
- : int list = [5; 4; 3; 2; 1]
```

Dans ces cas-là, il est toujours gênant de forcer l'utilisateur à passer un couple de liste dont le deuxième élément est toujours la liste vide. On écrira plutôt :

```
# let renverse = function l -> renverse_it (l,[]);;
val renverse : 'a list -> 'a list = <fun>
```

On pourra même se passer d'une définition préalable de `renverse_it`, qui n'a pas d'autre but que d'être utilisée par `renverse`, en définissant cette fonction utilitaire localement, grâce à l'instruction `let..in` (cf. 2.4.2)

```
# let renverse =
    function l ->
      let rec renverse_it =
        function   ([],resultat)         -> resultat
                 | (tete::suiv,resultat) -> renverse_it (suiv, tete::resultat)
      in
        renverse_it (l,[]);;
val renverse : 'a list -> 'a list = <fun>
```

Partitions

On se pose le problème de trouver tous les sous-ensembles d'un ensemble fini. On considère une liste dont les éléments sont tous différents comme étant un ensemble, et on définit la fonction `ssens` qui donne une liste de listes. Ces listes sont les sous-ensembles de la liste passée en argument.

Puis-je trouver les sous-ensembles de

```
tete::suiv
```

si je connais les sous-ensembles de `suiv`. La réponse est simple, les sous ensembles de

```
tete::suiv
```

sont ceux de `suiv`, ainsi que les sous-ensembles formés de l'ajout de `tete` aux sous-ensembles de `suiv`. On voit que les sous-ensembles de `suiv`, que l'on sait calculer, servent deux fois, et qu'il serait bon de ne les calculer qu'une seule fois. C'est un cas d'utilisation du `let...in` vu au paragraphe 2.4.2.

Disposant des sous-ensembles de `suiv`, qui est une liste de listes du genre `[[a;b]; [a]; ...]`, on aura à construire la liste des même listes avec `tete` en plus, à savoir `[[tete;a;b]; [tete;a];...]`. On utilisera pour cela la fonction `map` vue précédemment :

```
... map (function liste -> tete::liste) (ssens suiv) ...
```

La condition d'arrêt est que les sous-ensembles de l'ensemble vide sont un ensemble singleton, ne contenant comme élément que l'ensemble vide lui-même.

Allons-y donc :

```
# let rec ssens =
    function   []         -> [ [] ]
             | tete::suiv ->
      let ssens_suiv = ssens suiv
      in
         bout_a_bout ssens_suiv (map (function l -> tete::l) ssens_suiv);;
val ssens : 'a list -> 'a list list = <fun>
```

Voyez le type inféré ! On essaie ;

```
# ssens ['a';'b';'c';'d'];;
- : char list list =
[[]; ['d']; ['c']; ['c'; 'd']; ['b']; ['b'; 'd']; ['b'; 'c'];
 ['b'; 'c'; 'd']; ['a']; ['a'; 'd']; ['a'; 'c']; ['a'; 'c'; 'd'];
 ['a'; 'b']; ['a'; 'b'; 'd']; ['a'; 'b'; 'c']; ['a'; 'b'; 'c'; 'd']]
```

Chapitre 5

Mutabilité

Tout ce que l'on a vu jusqu'ici est basé sur la réécriture de *valeurs*, les valeurs étant des éléments d'un ensemble. Les valeurs ont quelque chose de figé : 1 c'est 1, ce ne sera jamais 2. Ce qui est moins figé, ce sont les symboles, dans la mesure où, en créant de nouvelles liaisons, on peut leur attribuer une valeur différente, c'est-à-dire une façon différente de les réécrire au moment d'une évaluation. Le concept de mutabilité désigne des valeurs qui peuvent « changer »... Donnons d'ores et déjà une vue imagée d'une valeur mutable. Une pièce de monnaie n'est pas mutable, une pièce de 1 euro, c'est toujours une pièce de 1 euro, alors que mon âge, change inexorablement tous les ans. Quel est le type d'une pièce, c'est-à-dire à quel ensemble une pièce donnée appartient ? Ce type est un sous-ensemble fini des décimaux, à savoir {0.01,0.02,0.05,0.1,0.2,0.5,1,2}. Quel est le type de mon âge ? On serait tenté de répondre entier, mais ça signifiera que mon âge est *une* valeur entière. C'est le cas pour le nombre de mes yeux, mais certainement pas pour mon âge. En fait, mon âge « fait référence » à une valeur entière à un moment donné, mais il fera référence à une autre valeur entière l'année prochaine. Le type de mon âge est donc non pas entier, mais « référence d'un entier ». Nous abordons cette notion de référence dans ce chapitre.

Avant de rentrer dans le vif du sujet, précisons qu'une donnée comme mon âge, dont « le contenu », « la référence », « la valeur » change est appelé une *variable* en informatique, et ce n'est pas la même chose qu'un symbole, qui lui n'est en fait qu'une règle de réécriture. Le hic, c'est que la notion de variable que vous connaissez déjà en mathématiques, c'est ce que nous désignons ici comme la notion de symbole (cf. la dicussion du paragraphe 2.1.2 page 5). Ne nous mégeanlons donc pas les cinpeaux.

5.1 Séquences

Avant d'entrer dans le vif de la mutabilité, faisons un préambule sur la notion de *séquence*. En `Caml` , on utilise le `;` pour séparer les éléments d'une liste, mais aussi pour séparer différentes évaluations. Ainsi, l'expression `exp1;exp2;exp3` se réécrit de la façon suivante : `Caml` évalue `exp1`, puis `exp2`, puis `exp3`. Le résultat de la dernière évaluation est ce en quoi se réécrit *toute* l'expression `exp1;exp2;exp3`. Les évaluations des expressions `exp1`, puis `exp2` semblent donc inutiles, nous verrons que pas nécessairement. Voici quelques exemples de réécriture de séquences d'expressions.

```
# let x=3;;
val x : int = 3
# let y="toto";;
val y : string = "toto"
# let z = true;;
val z : bool = true
# x;y;z ;;
- : bool = true
# z; x+32; y^" est content"; not z; "j'aime "^y;;
- : string = "j'aime toto"
```

Pour faire joli, et uniquement pour ça, `Caml` permet d'encadrer une séquence par les mots `begin` et `end`. On choisira aussi d'écrire chaque expression sur une ligne différente, pour plus de lisibilité.

```
# begin
      z;
      x+32;
      y^" est content";
      not z;
      "j'aime "^y;
  end;;
- : string = "j'aime toto"
```

Ce qu'il faut retenir, c'est que les séquences sont un mécanisme qui paraît bien inutile dans la mesure où on impose à `Caml` de faire plusieurs évaluations successives mais de ne garder que la dernière d'entre elles, qui elle seule est réécrite.

5.2 Les références

Le cas le plus simple de mutabilité est la référence. Soit un type donné, on peut définir une référence sur une valeur de ce type, référence que l'on pourra modifier ultérieurement. Nommons `i` une référence sur l'entier `38`.

```
# let i = ref 38;;
val i : int ref = {contents = 38}
```

`Caml` associe à `i` une valeur, de type `int ref`, qui est *une* référence sur un entier. Vérifions que `i` n'est pas un entier :

```
# i+1;;
This expression has type int ref but is here used with type int
```

Bien que cette incohérence de type soit logique, en pratique, on aimerait bien récupérer la valeur référée par la référence `i`, qui en ce moment est 38, pour en faire quelque chose. On utilise pour cela un `!`.

```
# !i;;
- : int = 38
# !i+1;;
- : int = 39
# i;;
- : int ref = {contents = 38}
```

Si les références sont appelés des *mutable*s, c'est parce qu'on peut justement modifier la valeur référée. Ici, sans refaire un `let i =...`, c'est-à-dire sans modifier la valeur de `i`, qui est une référence, on peut modifier la valeur entière que cette référence désigne. On utilise pour cela l'opérateur d'*affectation* `:=`.

```
# i:=72;;
- : unit = ()
# i;;
- : int ref = {contents = 72}
# !i;;
- : int = 72
```

Il est intéressant de remarquer que `Caml` réécrit l'expression `i:=72` en `()`, à savoir l'unique valeur du type `unit`[1]. En effet, une affectation est une modification de l'état de l'ordinateur, ce

1. Cf. 3.4 page 19.

qui n'a pas vraiment de raison de rendre une valeur particulière ! Certains langages, comme C , considèrent que la valeur d'une affectation est la valeur de ce qui est affecté. Dans ces langages, `i:=72` vaut `72`. En Caml , `i:=72` vaut `()`.

Regardons maintenant ce qui se passe lors de l'évaluation de la séquence suivante :

```
# begin
    i:=13;
    i:= !i+1;
    !i;
  end;;
- : int = 14
```

Caml commence par évaluer `i:=13`, ce qui se réécrit en `()` et modifie l'état de l'ordinateur de sorte que `i` désigne maintenant `13`. la valeur `()` sera perdue [2] car `i:=13` n'est pas la dernière expression de la séquence. Pour autant, cette modification n'a pas été inutile ! De même, l'expression `i:=!i+1` est évaluée en `()`, ce qui sera perdu, mais elle n'est pas vaine car son effet est de calculer la somme de la valeur référée par `i` [3], à savoir `!i`, à savoir `13`, de lui ajouter `1`, ce qui fait `14`, et d'affecter à la référence `i` la valeur `14` ainsi calculée. Enfin, la dernière évaluation, celle de l'expression `!i`, réécrit `14`, qui n'est pas perdu.

On voit sur cet exemple que *les séquences deviennent utiles dans la mesure où on travaille avec des mutables*. En poussant plus loin ce raisonnement, on peut dire qu'une séquence bien faite est constituée d'expressions qui, à part la dernière, doivent se réécrire en `()`, qui ne représente rien de particulier, et dont la perte n'est pas scandaleuse. Les concepteurs de Caml ont suivi ce raisonnement. Les exemples vus jusqu'à présent ont volontairement passé sous silence des messages d'alerte, voyons maintenant ce que répond effectivement Caml pour la séquence suivante :

```
# begin
    i:=13;
    "toto" ^ " est content";
    !i;
    i:= !i+1;
    38.6 +. 72.0;
    !i;
  end;;
Warning: this expression should have type unit.
Warning: this expression should have type unit.
Warning: this expression should have type unit.
- : int = 14
```

Les trois expressions qui ne sont pas les dernières de la séquence et qui de plus ne sont pas de type `unit` génèrent une alerte, signifiant au programmeur que « c'est louche ». Si on veut malgrer tout persister à mettre des expression qui ne sont pas de type `unit`, on peut dire à Caml « je sais ce que je fais, ne me mets pas un warning pour cette expression-là » en utilisant le mot-clé `ignore` :

```
# begin
    i:=13;
    ignore ("toto" ^ " est content");
    ignore (!i);
    i:= !i+1;
    ignore (38.6 +. 72.0);
    !i;
  end;;
- : int = 14
```

En langage parlé informatique, `i:=!i+1` se dit « *incrémentation* de la variable `i` ». Dans la plupart des langages, au lieu d'écrire `i:=!i+1`, on écrit `i = i+1`, ce qui ressemble à une absurdité

2. Perdre le résultat `()` n'est pas dramatique.
3. On dit parfois le contenu de `i`.

mathématique. En plus de cela, dans `i = i+1`, le `i` de gauche désigne la référence alors que le `i` de droite désigne la valeur référée. Dans ces autres langages, le symbole `i` désigne donc deux choses bien différentes. Dans ces langages, le même symbole `i` a une l-value [4], la référence, et une r-value [5], la valeur référée.

Cet aparté concernant des langages plus courants que `Caml` étant fait, on dira pour conclure qu'en `Caml` , un symbole est associé à une seule valeur, il n'y a pas de notion de l-value et r-value. Dans le cas d'une valeur mutable, comme `i` dans l'exemple précédent, la valeur associée à `i` est une référence sur une autre valeur, `!i`, et bien que la référence soit figée, comme toute valeur associée à un symbole, ce qu'elle réfère peut changer. Les notations `i` et `!i` en `Caml` lèvent toute ambiguïté entre la référence et ce qui est référé, et les deux n'ont pas le même type.

5.3 Structures de données

La notion de type en informatique correspond à la notion d'ensemble (cf. paragraphe 2.2 page 6), et on a vu que l'on pouvait construire un type comme l'union de deux autres types (cf. paragraphe 3.3 page 17), et également comme le produit cartésien de deux autres types (cf. paragraphe 3.2 page 16). Nous voyons dans cette section une autre façon de faire un produit cartésien sur les types. Cette façon n'implique pas la mutabilité, mais en pratique, on l'utilise beaucoup avec des types mutables, d'où l'introduction du concept de structure de données dans ce chapitre.

5.3.1 Généralités

Une *structure de données*, aussi appelée *type à champs nommés*, est un type qui du point de vue de la théorie des ensembles est analogue au produit cartésien. La différence est que l'accès aux composantes d'une valeur d'un type à champ nommé est plus simple, dans la mesure où ces composantes sont justement nommées.

Prenons l'exemple du type complexe. Les nombres complexes sont un agglomérat de deux réels, ce que l'on peut représenter par le type `float*float`. Soit $a = 2.5 + 3.8i$, on écrira [6] :

```
# type complex = Complexe of (float*float);;
type complex = Complexe of (float * float)
# let a = Complexe (2.5, 3.8);;
val a : complex = Complexe (2.5, 3.8)
```

Pour accéder à chacun des `float` agglomérés, il faut passer par un filtrage :

```
# let re = function Complexe (r,_) -> r;;
val re : complex -> float = <fun>
# let im = function Complexe (_,i) -> i;;
val im : complex -> float = <fun>
# re a;;
- : float = 2.5
# im a;;
- : float = 3.8
```

Un type structuré consiste à nommer les éléments du couple (ça s'étend bien sûr à un n-uplet quelconque), ce qui simplifie l'accès à ces éléments. Voici sur l'exemple des complexes comment on procède.

```
# type complexe = {re : float; im : float};;
type complexe = { re : float; im : float; }
# let a = {re = 2.5; im = 3.8};;
val a : complexe = {re = 2.5; im = 3.8}
# a.re;;
```

4. l comme left, à gauche de l'affectation.
5. r comme right, à droite de l'affectation.
6. La raison pour laquelle on utilise une étiquette est pour « forcer » `Caml` à utiliser notre type (cf. 3.1 page 16).

```
- : float = 2.5
# a.im;;
- : float = 3.8
```

Dans cet exemple, la déclaration de type dit qu'un complexe est un agglomérat de deux floats, le premier s'appellent `re` et le deuxième `im`. Une valeur de ce type a la forme `{ re = ... ; im = ... }`, et l'ordre n'a pas d'importance. Pas besoin de filtrage pour accéder aux composants, il suffit d'utiliser le `.` et le tour est joué.

Autre exemple très classique :

```
# type humain = {
    nom : string;
    prenom: string;
    nb_yeux : int
  };;
type humain = { nom : string; prenom : string; nb_yeux : int; }
# let auteur_poly = {
    nom     = "Frezza-Buet";
    prenom  = "Hervé";
    nb_yeux = 2
  };;
val auteur_poly : humain =
  {nom = "Frezza-Buet"; prenom = "Hervé"; nb_yeux = 2}
```

5.3.2 Structures de données et mutabilité

Comme discuté dans l'exemple au début de ce chapitre, l'âge d'un humain doit être mutable, contrairement à son nom, son prénom, et son nombre d'yeux [7]. Ajoutons donc un champ mutable, et faisons viellir l'auteur du poly d'un an.

```
# type humain = {
    nom : string;
    prenom: string;
    nb_yeux : int;
    age : int ref;
  };;
type humain = {nom : string; prenom : string; nb_yeux : int; age : int ref}
# let auteur_poly = {
    nb_yeux    = 2;
    prenom     = "Hervé";
    age        = ref 30;
    nom        = "Frezza-Buet"
  };;
val auteur_poly : humain =
  {nom = "Frezza-Buet"; prenom = "Hervé"; nb_yeux = 2; age = {contents = 30}}

# auteur_poly.age := !(auteur_poly.age) +1;;
- : unit = ()
# auteur_poly;;
- : humain = {nom = "Frezza-Buet"; prenom = "Hervé";
              nb_yeux = 2; age = {contents = 31}}
```

Comme il est très fréquent d'avoir des champs mutables dans un type à champs nommés, la syntaxe de `Caml` fournit des facilités pour déclarer ces champs. On utilisera le mot-clé `mutable`, et l'opérateur `<-` pour l'affectation. Néanmoins, le principe de fond ne change pas par rapport à l'exemple précédent. Nous ne fesons que l'écrire à nouveau, en profitant des facilités syntaxiques de `Caml` :

7. que nous aurons l'optimisme de supposer constant.

```
# type humain = {
            nom     : string;
            prenom  : string;
            nb_yeux : int;
    mutable age     : int;
  };;
type humain = {
  nom : string;
  prenom : string;
  nb_yeux : int;
  mutable age : int;
}
# let auteur_poly = {
      nb_yeux    = 2;
      prenom     = "Hervé";
      age        = 30;
      nom        = "Frezza-Buet";
  };;
val auteur_poly : humain =
  {nom = "Frezza-Buet"; prenom = "Hervé"; nb_yeux = 2; age = 30}
# auteur_poly.age <- auteur_poly.age+1;;
- : unit = ()
# auteur_poly;;
- : humain = {nom = "Frezza-Buet"; prenom = "Hervé"; nb_yeux = 2; age = 31}
```

On constate que la syntaxe de `Caml` allège l'écriture. En effet, une fois qu'un champ est déclaré `mutable`, il n'est plus utile de faire apparaître les `ref`. On retrouve alors une ambiguïté[8] du symbole `auteur_poly.age` dans l'expression `auteur_poly.age <- auteur_poly.age+1`. A gauche de `<-`, il s'agit de la référence, dans tous les autres cas, de la valeur référée.

5.4 Les tableaux

Un *tableau* est un type mutable très utilisé en informatique. En fait, ce n'est pas directement un type, mais un concept au-dessus, comme le sont les listes. Reportez-vous au paragraphe 3.7 page 26 pour voir ce qu'on entend ici par « concept au-dessus ».

Un tableau est grossièrement analogue à une liste de mutables, tous de même type. La différence est que le nombre d'éléments de cette liste est déterminé une bonne fois pour toutes, et l'accès à ces éléments peut se faire directement, sans parcourrir la liste, en précisant simplement son rang[9].

Alors que les listes sont notées avec des `[..]`, les tableaux sont notés, eux, avec des `[|...|]`. On retrouve également l'opérateur `<-` pour l'affectation, un masquage de la notion de référence comme pour les types à champs nommés, et donc l'ambiguité du symbole associé à un élément du tableau, comme nous l'avons déjà discuté page 40.

Découvrons donc les tableaux via un petit exemple :

```
# let tab = [| "vert" ; "bleu" ; "rouge" ; "cassoulet" |];;
val tab : string array = [|"vert"; "bleu"; "rouge"; "cassoulet"|]
# tab.(0);;
- : string = "vert"
# tab.(3);;
- : string = "cassoulet"
# tab.(1)<-" toulousain";;
- : unit = ()
# tab;;
- : string array = [|"vert"; " toulousain"; "rouge"; "cassoulet"|]
```

8. Cf. page 40.
9. Le numéro de sa place, le premier étant 0.

```
# tab.(3)<-tab.(3)^tab.(1);;
- : unit = ()
# tab;;
- : string array = [|"vert"; "toulousain"; "rouge"; "cassoulet toulousain"|]
```

Le numéro de l'élément, exprimé entre `.()`, est appelé l'*indice* de l'élément. Souvent, et c'est là que c'est intéressant, c'est le résultat d'un calcul qui sert d'indice.

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# tab.(!i -1) ^ tab.(!i +1);;
- : string = "vertrouge"
# i:=!i+1;;
- : unit = ()
# tab.(!i -1) ^ tab.(!i +1);;
- : string = " toulousaincassoulet toulousain"
```

Enfin, précisons qu'en pratique, les tableaux peuvent etre de très grande taille, ce qui exclut de les définir en utilisant `let tab = [| ... |]`. On utilisera alors la fonction `Array.create`[10], qui construit un tableau contenant des éléments identiques. De même, la fonction `Array.length` donne la taille du tableau.

```
# let a = Array.create 10 7.32;;
val a : float array =
  [|7.32; 7.32; 7.32; 7.32; 7.32; 7.32; 7.32; 7.32; 7.32; 7.32|]
# Array.length a;;
- : int = 10
```

5.5 Boucles

Nous avons vu que la notion de mutabilité a amené l'utilisation de séquences, ce qui nous éloigne un peu du modèle de réécriture d'une formule mathématique, dont nous étions partis au chapitre 2. En effet, à partir du moment où on modifie des états internes de l'ordinateur, les valeurs référées, et qu'on fait ces modifications tour à tour, on passe d'une approche « réécriture » à une approche « recette de cuisine » de la programmation. Il s'avère qu'on maîtrise beaucoup moins la cuisine que les mathématiques, ce qui a pour conséquence l'apparition de nombreux bugs en même temps que la mutabilité ! Il n'empêche que les mutables sont très utilisés et ont leur intérêt, mais vous ne direz pas qu'on ne vous avait pas prévenu...

Dans les meilleures recettes de cuisine, comme par exemple celle de la pizza, il y a toujours un moment où il faut surveiller les opérations, moment qui est exprimé par une phrase du genre « tant que la pâte n'est pas élastique, pétrissez énergiquement ». On appelle ce genre de phrase une *boucle* en informatique, ce qui s'illustre par le dessin de la figure 5.1. Ce type de dessin est appelé organigramme, et est aujourd'hui tombé en désuétude... sauf dans le chapitre « boucle » des cours d'informatique.

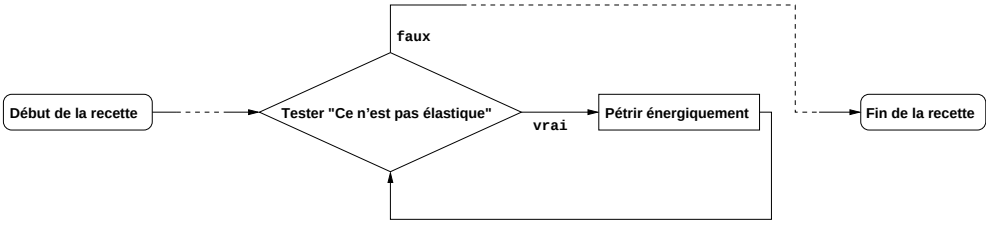

FIGURE 5.1 – Boucle pour pétrir la pâte à pizza. Suivez les flèches...

10. Le `.` dans le nom de fonction `Array.create` peut vous paraître étrange. Il est dû au fait que `Caml` est un langage objet, ce qui dépasse le cadre de ce que nous abordons ici.

5.5.1 La boucle `while`

Le cas de la pâte à pizza est représentatif de l'ensemble des boucles en informatique. On peut en effet toujours se ramener à « tant que `A`, faire `B` ». L'expression `A` est un test, donc une valeur de type `bool`, et `B` est une expression, de type `unit` puisqu'elle risque d'être répétée, ce qui lui confère un statut analogue aux expressions d'une sequence qui ne sont pas la dernière. Cela se traduit en `Caml` par l'expression

```
while A do B done
```

qui correspond à la recette de cuisine de la figure 5.2.

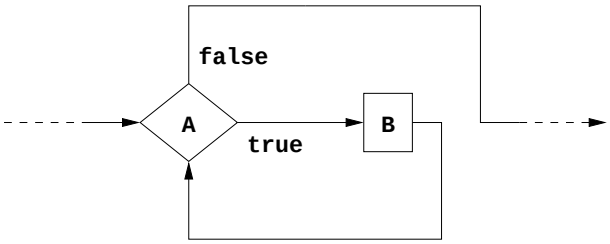

FIGURE 5.2 – La boucle `while A do B done`.

Exécutez à la main l'évaluation des expressions suivantes pour constater que `Caml` évalue l'expression comme la somme des éléments du tableau.

```
# let tab = [| 12; 25; 53; 6; 4 |];;
# let i = ref 0;;
# let s = ref 0;;
# begin
    while !i <= ((Array.length tab) -1) do
      s := !s + tab.(!i);
      i := !i + 1;
    done;
    !s;
  end;;
- : int = 100
```

5.5.2 La boucle `for`

Une boucle `for` n'est rien d'autre qu'un cas particulier de boucle `while`. En effet, souvent, une boucle consiste à faire varier une référence entière (`i` dans l'exemple précédent) d'une valeur initiale jusqu'à une valeur finale. Cette valeur sert souvent d'indice à un tableau. On réécrit l'exemple précédent à l'aide de la boucle `for`, ce qui ne change rien à part la syntaxe, et le fait que le symbole `i` est pris en charge par la syntaxe : il ne faut plus le définir avec un `let`, et son type est `int`, et non `int ref`. En fait, le `i` qui apparaît dans la boucle for est la valeur référée d'un `int ref` qui nous est caché.

```
# let tab = [| 12; 25; 53; 6; 4 |];;
# let s = ref 0;;
# begin
    for i=0 to (Array.length tab -1) do
      s := !s + tab.(i);
    done;
    !s;
  end;;
- : int = 100
```

5.6 Le branchement

Bien que ce ne soit pas strictement lié à la notion de mutabilité, la notion de *branchement* se retrouvre fréquemment dans les recettes de cuisine, et nous l'abordons dans ce chapitre. Exemple de ce qu'on appelle un branchement : « Si il y a des lardons dans la sauce, alors ne salez pas, sinon, ajoutez une pincée de sel ». La même chose exprimée en `Caml` donne :

```
if A then B else C
```

Si `A` se réécrit en `true`, l'expression se réécrit en `B`, sinon, elle se réécrit en `C`. Notez qu'on peut se passer du `else C` dans le cas où `B` est de type `unit`.

5.7 Exemples d'utilisation

5.7.1 Tableaux et boucles

Ecrivons une fonction qui donne la plus grande valeur contenue dans un tableau. On supposera que `comp` est une fonction qui prend deux éléments et qui retourne un booléen, qui est vrai si le premier argument est supérieur où égal au second. On initialisera une référence nommée `max` sur le premier élément du tableau, et on testera tous les autres. Si l'un d'eux est supérieur à ce qu'on pense etre le max, au vu de l'examen des éléments précédents du tableau, on met à jour le max.

```
# let maximum = function tab ->
                function comp  ->
    let     max = ref tab.(0)
    in let  sup = (Array.length tab) - 1
    in begin
         for i=0 to sup do
           if (comp (tab.(i),(!max)))
           then
             max := tab.(i)
         done;
         !max; (* c'est le résultat *)
       end;;
```

On essaie :

```
# maximum [| 1; 38; 12; 4 |] (function (x,y) -> x >= y);;
- : int = 38
```

On peut l'appliquer aussi, grâce au polymorphisme, à un tableau de listes, pour savoir laquelle est la plus grande.

```
# let rec est_plus_grande =
    function   ([], []) -> true
             | (_ , []) -> true
             | ([], _)  -> false
     | (_::r1, _::r2) -> (est_plus_grande (r1,r2));;
val est_plus_grande : 'a list * 'b list -> bool = <fun>
# maximum [|
             [ "toto"; "titi"; "tutu" ];
             [];
             [ "je"; "suis"; "la"; "plus"; "grande"; "liste" ];
             [ "je ne suis pas la plus grande liste, mais j'en ai l'air" ]
          |] est_plus_grande;;
- : string list = ["je"; "suis"; "la"; "plus"; "grande"; "liste"]
```

5.7.2 Champs fonctionnels

Nous allons ici illustrer, au sujet des types à champs nommés, le fait qu'il peut être très pratique d'avoir une valeur fonctionnelle comme champ. Par exemple, lorsque plusieurs choses sont différentes, mais qu'elles savent toute réaliser, à leur façon, une même opération, cette opération peut-être stockée comme un champ fonctionnel. Prenons un exemple. Les mammifères sont de différentes sortes (vaches, cochon, etc...) mais ils savent tous produire un bruit quand on les pince : ils crient. Vu comme un problème informatique, le cas des mammifère peut s'exprimer comme suit : « Bien qu'ils soient tous différents, quand je vois un mammifère, je peux compter sur lui pour me donner une fonction `crie` ». Cette fonction prendra en argument un entier, d'autant plus élevé qu'on pince fort, et retournera une chaîne de caractères correspondant au cri produit. Définissons donc le type `mammifere` :

```
# type mammifere = {
    nom    : string;
    espece : string;
    crie   : int -> string;
  };;
```

Avant de rentrer dans le vif du sujet, définissons quelques fonctions de cri intermédiaires.

```
# let rec crie_humain =
    function   0 -> ""
             | 1 -> "Aie."
             | n -> "Aie " ^ (crie_humain (n-1));;
val crie_humain : int -> string = <fun>
# crie_humain 4;;
- : string = "Aie Aie Aie Aie."
# let crie_chevre =
    function n ->
      let rec e = function   0 -> ""
                           | n -> "e" ^ (e (n-1))
      in
        "B" ^ (e n) ^ "!";;
val crie_chevre : int -> string = <fun>
# crie_chevre 7;;
- : string = "Beeeeeee!"
# let crie_cochon = function n -> "Gruik!";;
val crie_cochon : 'a -> string = <fun>
# crie_cochon 100;;
- : string = "Gruik!"
```

Définissons maintenant des fonctions permettant de créer simplement des mammifères :

```
# let new_humain =
    function blaze ->
      {
         nom    = blaze;
         espece = "humain";
         crie   = crie_humain;
      };;
val new_humain : string -> mammifere = <fun>
# new_humain "Jean-Claude";;
- : mammifere = {nom = "Jean-Claude"; espece = "humain"; crie = <fun>}
# let new_chevre =
    function blaze ->
      {
         nom    = blaze;
         espece = "chevre";
```

```
         crie   = crie_chevre;
      };;
# let new_cochon =
    function blaze ->
      {
         nom    = blaze;
         espece = "cochon";
         crie   = crie_cochon;
      };;
```

Construisons maintenant une fonction `pince_liste` qui prend en argument un entier (la force du pincement initial) puis une liste de mammifères. Elle retourne la liste des cris obtenus en pinçant tour à tour ces mammifères de plus en plus fort. Comme ils savent tous crier, *je n'ai pas besoin de savoir quelle sorte de mammifère ils sont.*

```
# let rec pince_list =
    function n ->
    function   []   -> []
             | t::r -> (t.crie n) :: (pince_list (n+1) r);;
val pince_list : int -> mammifere list -> string list = <fun>
```

Notez bien la forme `t.crie` de la fonction « portée par le mammifère `t` ». On retrouvera cette forme dans les langages à objets, dans un autre cadre que celui de ce document. Revenons à nos moutons, et essayons de pincer toute une ferme, avec un pincement initial de 2.

```
# let ferme = [
    new_humain "Mr Dufermier";
    new_humain "Mme Dufermier";
    new_cochon "Bebert";
    new_chevre "Blanquette";
    new_chevre "Biquette";
    new_cochon "Porcinet" ];;
val ferme : mammifere list =
  [{nom = "Mr Dufermier"; espece = "humain"; crie = <fun>};
   {nom = "Mme Dufermier"; espece = "humain"; crie = <fun>};
   {nom = "Bebert"; espece = "cochon"; crie = <fun>};
   {nom = "Blanquette"; espece = "chevre"; crie = <fun>};
   {nom = "Biquette"; espece = "chevre"; crie = <fun>};
   {nom = "Porcinet"; espece = "cochon"; crie = <fun>}]
# pince_list 2 ferme;;
- : string list =
["Aie Aie."; "Aie Aie Aie."; "Gruik!"; "Beeeee!"; "Beeeeee!"; "Gruik!"]
```

Chaque mammifère réagit donc à sa façon, et le malicieux pinceur, lui, quand il écrit la fonction `pince_list`, ne se soucie pas de savoir comment chaque mammifère va réagir. C'est la toute la terrible morale de ce conte bucolique.

5.7.3 Fermeture et mutabilité

Le but de cet exemple est de présenter un des intérêt de la notion de *fermeture lexicale*[11] lorsqu'on utilise des mutables. On a vu[12] que l'on peut réaliser l'application partielle d'une fonction à plusieurs arguments. Dans ce cas, `Caml` calcule une fonction qui « attend les arguments manquants », ayant stocké dans la fermeture lexicale la valeur des arguments présents. Définissons un porte monnaie comme une fonction à deux arguments, le premier étant une référence sur un `float` qui est la valeur de ce que contient le porte-monnaie, et le deuxième une action à réaliser sur le porte-monnaie. Définissons d'abord un type pour décrire les actions possibles sur le porte-monnaie.

11. Cf. paragraphe 2.5.1 page 9.
12. Cf. paragraphe 2.5.3 page 12.

```
# type action =   VIDE            (* vider le contenu      *)
                | AJOUTE of float (* Mettre des sous       *)
                | RETIRE of float (* Retirer des sous      *)
                | CONTENU;;       (* Regarder combien on a *)
```

Définissons un type pour le résultat de ces actions. Les actions `AJOUTE` et `RETIRE` modifient le porte-monnaie, mais ne fournissent rien comme résultat, ce qu'on notera `FAIT`. Les actions `VIDE` et `CONTENU` donnent un `float` comme résultat. D'où le type des résultats possibles des actions :

```
# type resultat =   FAIT
                  | VALEUR of float;;
```

L'idée est de définir une fonction `porte_monnaie` qui prend deux arguments, une référence sur un `float` et une action. Cela donnera quelque chose comme :

```
# let porte_monnaie =
   function contenu ->  (* la reference *)
   function   VIDE           -> ...
            | AJOUTE montant -> ...
            | RETIRE montant -> ...
            | CONTENU        -> ...
```

En pratique, on utilisera cette fonction comme suit :

```
# let bourse = porte_monnaie (ref 0.0);;
```

On a alors `bourse` qui est l'application de `porte_monnaie` au premier de ses arguments uniquement. `bourse` est donc une fonction, dont la fermeture contient la valeur du premier argument, à savoir une référence, et qui attend une action pour l'exécuter sur cette référence. Comme la fonction `porte_monnaie` a vocation à ce qu'on lui applique partiellement son premier argument, on utilisera la syntaxe suivante pour sa définition, au lieu de celle que nous avons esquissée et que nous connaissons déjà. Cette nouvelle écriture consiste à passer `function contenu ->` à gauche du `=`, en éliminant le `function ... ->`. Cela dit, cette nouvelle syntaxe est équivalente à l'autre, et les deux fonctionnent.

```
# let porte_monnaie contenu =
   function   VIDE           -> let valeur = !contenu  (* Sauvegarde *)
                                in begin
                                       (* On vide le porte monnaie,
                                          et on retourne sa valeur *)
                                    contenu := 0.0;
                                    VALEUR valeur;
                                      (* Heureusement qu'on l'avait
                                  sauvegardee avant d'annuler
                                          le contenu ! *)
                                   end
            | AJOUTE montant -> begin
                                  contenu := !contenu +. montant;
                                  FAIT;
                                end
            | RETIRE montant -> begin
                                  contenu := !contenu -. montant;
                                  FAIT;
                                end
            | CONTENU        -> VALEUR (!contenu);;
val porte_monnaie : float ref -> action -> resultat = <fun>
```

Créons maintenant un porte-monnaie, et utilisons-le.

```
# let bourse = porte_monnaie (ref 0.0);;
val bourse : action -> resultat = <fun>
# bourse;;
- : action -> resultat = <fun>
# bourse (AJOUTE 10.25);;
- : resultat = FAIT
# bourse (RETIRE 2.10);;
- : resultat = FAIT;;
# bourse CONTENU;;
- : resultat = VALEUR 8.15
# bourse VIDE;;
- : resultat = VALEUR 8.15
# bourse CONTENU;;
- : resultat = VALEUR 0
```

Chapitre 6

Conclusion

On peut multiplier les exemples où l'approche fonctionnelle, surtout lorsqu'elle est typée, permet de programmer plus sûrement, avec des programmes récursifs qui expriment des traitements complexes avec concision. La perte de performance, en termes de temps d'exécution, n'est que rarement rédhibitoire. De plus, l'approche fonctionnelle est également utilisée dans un domaine proche, celui du calcul formel. Ainsi, les principes de programmation décrits ici sont réutilisables dans d'autres cadres que l'informatique à proprement parler, et peuvent servir de base aux physiciens, chimistes, qui utilisent les calculateurs formels comme outil.

Il faut deuxièmement retenir que l'on peut faire de la programmation fonctionnelle dans quasiment tous les langages, plus ou moins facilement, et en perdant en général le garde fou qu'est l'inférence de type. Avoir une vue `Caml` de ce type de programmation incite à « faire propre malgré tout » dans des langages de plus bas niveau, comme `C`, ce qui augmente la qualité et la maintenabilité du code.

Enfin, il est à noter que dès qu'apparaît la notion de mutabilité, un cortège de vérification de dépassement d'indices des tableaux, de variables mal initialisées, de boucles infinies et bien d'autres sources de bugs ne manque pas de défiler avec une arrogance intolérable. Bien que la chasse aux bugs soit le passe-temps dominical le plus pratiqué par les informaticiens, elle n'en reste pas moins une activité peu gratifiante qui tend à discréditer la profession. Soyez acharnés, mais discrets...

Bibliographie

[Paulson, 1991] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge university press, 1991.

[Wikström, 1987] Ake Wikström. *Functional Programming Using Standard ML*. Prentice Hall international series in computer science. Prentice-Hall, 1987.

Index