Plus de précisions sur les motifs

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, **`true`**, `()`, `'a'`, `"abc"`, *etc.*) ;
2. les motifs **à paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers ;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

– Exemple –

```
let somme p =
    match p with
        |Droite x -> x
        |Plan (x, y) -> x + y
        |Espace (x, y, z) -> x + y + z
```

Les noms dans les motifs ne font pas référence à leur définition passée.

– Exemple –

```
let f x =
    let n = 2 in
    match x with
        |0 -> 0
        |n -> -1
        |_ -> 1
```

```
# f 0;;
- : int = 0
# f 3;;
- : int = -1
```

Le `n` du 2e motif ne fait pas référence à la liaison précédente. Ce motif `n` filtre tout.

Exemple : évaluation de formules — type formule

Nous souhaitons représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une formule est une donnée récursive :

1. c'est un atome P ;
2. ou bien est la négation d'une formule $(\neg F)$;
3. ou bien est la conjonction de deux formules $(F \wedge G)$;
4. ou bien est la disjonction de deux formules $(F \vee G)$.

On en déduit la définition de type (somme à paramètres et récursive) suivante :

```
type formule =
    |Atome of char
    |Non of formule
    |Et of formule * formule
    |Ou of formule * formule
```

Exemple : évaluation de formules — valuations et évaluation

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool
```

La fonction d'**évaluation** d'une formule sous une valuation s'écrit très simplement au moyen d'un filtrage :

```
let rec evaluer form valu =
    match form with
        |Atome c -> valu c
        |Non f -> not (evaluer f valu)
        |Et (f, g) -> (evaluer f valu) && (evaluer g valu)
        |Ou (f, g) -> (evaluer f valu) || (evaluer g valu)
```

Exemple : évaluation de formules

– Exemple –

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation v

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela, f est codée par

```
let f =
    Et (
        (Non (Atome 'P')),
        (Ou (Atome 'P', Atome 'R'))
    )
```

et v par

```
let v c =
    match c with
        |'P' -> false
        |'R' -> true
        |_ -> false
```

L'évaluation de f sous v donne

```
# evaluer f v;;
- : bool = true
```

Plan

6. Notions

Définition

Une fonction d'ordre supérieur est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un **paramètre** de type fonction ;
2. `f` **renvoie** une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement **générique**.

Le fait de pouvoir renvoyer une fonction est un procédé très puissant en programmation fonctionnelle. Le programmeur n'est plus le seul concepteur de fonctions : l'exécution / l'interprétation peut en **créer à la volée** et en appeler.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

`E1 -> E2 -> ... -> En -> S`,

toute **application partielle**

`f e1 e2 ... ek`

avec `1 ⩽ k ⩽ n - 1` produit une valeur de type

`Ek+1 -> ... -> En -> S`

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont curryfiées en Caml : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions** (qui admettent donc un type de la forme `E1 -> (E2 -> ... -> En -> S)`).

On peut donc voir toute fonction à deux paramètres ou plus comme une fonction d'ordre supérieur car son application partielle renvoie une fonction.

Fonctions renvoyant des fonctions

– Exemple –

Analysons le type de la fonction

```
let encadrer u w =
    fun v -> u ^  v ^  w
```

On infère le type

`string -> string -> string -> string`.

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string)`,

Ceci montre que `encadrer` renvoie une fonction de type `string -> string`. C'est donc une fonction d'ordre supérieur.

L'appel `encadrer u v` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = encadrer "aa" "bb";;
val f : string -> string = <fun>
```

```
# f "bab";;
- : string = "aababbb"
```

Fonctions paramétrées par des fonctions

– Exemple –

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        f (appli_repetee f x (n - 1))
```

On infère le type

`(int -> int) -> int -> int -> int`,

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.
Elle calcule l'application de la composée n^e de `f` sur l'entier `x`, c.-à-d.,

$$f^n(x)$$

```
# appli_repetee (fun x -> x + 1) 3 4;;
- : int = 7
```

```
# appli_repetee (fun x -> 2 * x) 3 4;;
- : int = 48
```

Fonctions paramétrées par et renvoyant des fonctions

– Exemple –

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

```
(int -> int) -> (int -> int) -> int -> int
```

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `entrelacer f g` renvoie une fonction qui accepte un entier `x` et renvoie

$$\texttt{f}\,(\texttt{g}\,(\texttt{f}\,(\texttt{g}\,(\texttt{x})))).$$

```
# let h = entrelacer (fun x -> x * 2) (fun x -> x + 1);;
val h : int -> int = <fun>
```

```
# h 3;;
- : int = 18
```

Exemple complet 1 : mots fonctionnels

Reprenons la façon fonctionnelle de représenter les mots vue précédemment :

```
type 'a mot = {
    lettres : int -> 'a;
    longueur : int
}
```

On rappelle que si `u` est un mot, l'expression `u.lettres i` a pour valeur la `i`^{e} lettre de `u`.

– Exemple –

```
let mot_3 = {
    lettres =
    (fun i ->
        match i with
            |1 -> 'a'
            |2 -> 'b'
            |_ -> 'b'
    );
    longueur = 3
}
```

représente le mot (de lettres de type `char`) `abb`.

Exemple complet 1 : mots fonctionnels

La fonction ci-contre **convertit** un mot en une **chaîne de caractères** qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = String.make 1 (lettre_vers_char (u.lettres i)) in
            ch ^  (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string` ;
- `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char` ;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

– Exemples –

```
# string_of_mot mot_3 (fun x -> x);;
- : string = "abb"
```

```
# string_of_mot {lettres = (fun _ -> true); longueur = 4}
    (fun b -> if b then "T" else "F");;
- : string = "TTTT"
```

Exemple complet 1 : mots fonctionnels

La fonction ci-contre **concatène** les deux mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

– Exemple –

```
# let mot_4 = concatener mot_3 mot_3 in
  string_of_mot mot_4 (fun x -> x);;
- : string = "abbabb"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geqslant 0$, le n^e mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1}f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun _ -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun _ -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

Avec la liaison préalable `# let id = fun x -> x;;`,

```
# let w = mot_fibo 0 in string_of_mot w id;;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w id;;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w id;;
- : string = "ab"
```

```
# let w = mot_fibo 3 in string_of_mot w id;;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w id;;
- : string = "abaab"
# let w = mot_fibo 5 in string_of_mot w id;;
- : string = "abaababa"
```

Exemple complet 2 : images fonctionnelles

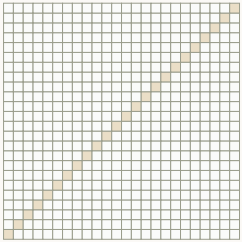
Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

– Exemple –

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y then
            {rouge = 127; vert = 127; bleu = 127}
        else
            {rouge = 255; vert = 255; bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

représente l'image



Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en **complémentant les couleurs** des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
        vert = 255 - (im.contenus_pixels p).vert;
        bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels; largeur = im.largeur; hauteur = im.hauteur}
```

Voici une fonction qui calcule l'**image miroir** d'une image en entrée :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x + 1, y)
    in
    {contenus_pixels = contenus_pixels; largeur = im.largeur; hauteur = im.hauteur}
```

Ces fonctions s'évaluent en temps $\Theta(1)$. La complexité ne dépend donc pas de la taille de l'image !

Exemple complet 3 : séries génératrices

On souhaite représenter des séries génératrices. Ce sont des polynômes en une variable t de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n t^n$$

où les α_i sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de **coder** de manière compacte des **suites infinies d'entiers**

$$(\alpha_0, \alpha_1, \alpha_2, \ldots).$$

– Exemple –

La série génératrice de la suite $(1, 2, 4, 8, 16, \ldots)$ des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \cdots .$$

Question : comment représenter des séries génératrices ?

Exemple complet 3 : séries génératrices

Réponse : par une **fonction** qui à tout entier positif n associe le coefficient α_n de t^n.

Ceci est offert par le type

```
type serie_gen = int -> int
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère **infini** de ces objets (degré possiblement infini).

– Exemple –

La série génératrice des puissances de 2 est ainsi codée par

```
let puissances_2 =
    fun n -> int_of_float (2. ** (float_of_int n))
```

Exemple complet 3 : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

1. somme :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right) + \left(\sum_{m\geqslant 0}\beta_m t^m\right) = \sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\,;$$
2. produit d'Hadamard :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right) \boxtimes \left(\sum_{m\geqslant 0}\beta_m t^m\right) = \sum_{k\geqslant 0}\alpha_k\beta_k t^k\,;$$
3. produit :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right) \cdot \left(\sum_{m\geqslant 0}\beta_m t^m\right) = \sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}t^k.$$

Il est possible de les implanter simplement en utilisant des fonctions d'ordre supérieur.

Exemple complet 3 : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =
    fun k -> (s1 k) + (s2 k)
```

De manière équivalente, sans fonction anonyme :

```
let somme s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res
```

– Exemple –

```
# let s3 = somme puissances_2 puissances_2;;
val s3 : int -> int = <fun>
```

```
# s3 3;;
- : int = 16
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =
    fun k -> (s1 k) * (s2 k)
```

Exemple complet 3 : séries génératrices

L'implantation du produit est un peu plus technique dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let produit s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then 0
            else (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        aux 0
    in
    resultat
```

– Exemples –

```
# let sg_un = fun n -> 1;;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = produit sg_un sg_un;;
val sg_un_carre : int -> int = <fun>

# (sg_un_carre 0), (sg_un_carre 1), (sg_un_carre 2), (sg_un_carre 3);;
- : int * int * int * int = (1, 2, 3, 4)
```

6. Notions

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

1. Une fonction polymorphe est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

– Exemple –

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

2. Un type polymorphe est un type paramétré.

– Exemple –

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

3. Une valeur polymorphe est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé.

– Exemple –

```
# Vide;;
- : 'a liste = Vide
```

Polymorphisme paramétrique

En Caml, le polymorphisme est paramétrique : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« *tout ou un* »).

De cette manière, pour **déterminer le type d'un paramètre** `x` d'une fonction correctement typée, le système de typage fonctionne (de manière très simplifiée) ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de `x`), alors `x` est du type le plus général `'a`.

Fonctions standard polymorphes

Beaucoup de fonctions standard sont polymorphes. Parmi elles :

- `( = ) : 'a -> 'a -> bool` teste l'égalité entre deux valeurs d'un même type.
- `( <> ) : 'a -> 'a -> bool` teste la différence entre deux valeurs d'un même type.
- `compare : 'a -> 'a -> int` Renvoie `-1` (resp. `1`) si la 1re valeur est strictement inférieure (resp. supérieure) à la 2e et `0` sinon.

  Ceci est une fonction générique de comparaison entre deux valeurs d'un même type.
- `fst : 'a * 'b -> 'a` Renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
- `snd : 'a * 'b -> 'b` Renvoie la 2e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.

Exemple : exponentiation rapide

Le calcul de x^n, où x est un entier, se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k + 1). \end{cases}$$

Ceci se traduit en

```
let rec puiss x n =
    if n = 0 then
        1
    else
        let tmp = puiss x (n / 2) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

Exemple : exponentiation rapide

L'opération d'exponentiation peut s'appliquer à d'**autres objets** que des entiers, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

(En d'autres termes, ces objets doivent former une structure de monoïde.)

On souhaite ainsi écrire une fonction polymorphe `puiss_poly` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `('e -> 'e -> 'e)` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;
- Le type de retour est `'e`.

La valeur renvoyée est x^n.

Exemple : exponentiation rapide

On obtient ainsi la **fonction polymorphe**

```
let rec puiss_poly op unite x n =
    if n = 0 then
        unite
    else
        let tmp = puiss_poly op unite x (n / 2) in
        if n mod 2 = 0 then
            op tmp tmp
        else
            op (op tmp tmp) x
```

– Exemples –

```
# puiss_poly (fun a b -> a + b) 0 1 6;;
- : int = 6
# puiss_poly (fun a b -> a * b) 1 2 10;;
- : int = 1024
# puiss_poly (fun u v -> u ^v) "" "abb" 4;;
- : string = "abbabbabbabb"
# puiss_poly (fun a b -> a || b) false false 293898273;;
- : bool = false
```

Exemple : exponentiation rapide

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Pour cela, on représente les monoïdes au moyen du type enregistrement

```
type 'a monoide = {
    op : 'a -> 'a -> 'a;
    unite : 'a
}
```

La fonction précédente devient

```
let rec puiss_poly monoide x n =
    if n = 0 then
        monoide.unite
    else
        let tmp = puiss_poly monoide x (n / 2) in
        if n mod 2 = 0 then
            monoide.op tmp tmp
        else
            monoide.op (monoide.op tmp tmp) x
```