12.2. Formules logiques

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. il s'agit d'un atome p ;
2. ou bien de la négation d'une formule $(\neg F)$;
3. ou bien de la conjonction de deux formules $(F_1 \wedge F_2)$;
4. ou bien de la disjonction de deux formules $(F_1 \vee F_2)$.

Ceci se traduit en le type récursif

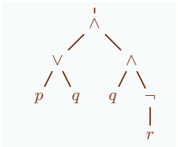
```
type formules =
    |Atome of char
    |Non of formules
    |Et of formules * formules
    |Ou of formules * formules
```

Exemple

La formule logique $(p \vee q) \wedge (q \wedge (\neg r))$ est représentée par l'expression

```
Et (Ou ((Atome 'p'), (Atome 'q') ), Et ((Atome 'q'), (Non (Atome 'r'))))
```

Il s'agit également de l'arbre



Nous souhaitons maintenant écrire une fonction permettant d'**évaluer** une formule.

Évaluer une formule se fait toujours dans le contexte d'une *valuation* donnée : elle informe sur la valeur de vérité (« vrai » ou « faux ») prise par chacun des atomes de la formule. Ces valeurs de vérité sont représentées par le type

```
type valeurs_verite = Faux | Vrai
```

Exemple

Une valuation possible (parmi d'autres) de la formule $(p \vee q) \wedge (q \wedge (\neg r))$ est

$$p \mapsto \text{vrai}, \quad q \mapsto \text{faux}, \quad r \mapsto \text{vrai}.$$

Il y en a $2^3 = 8$ différentes car chaque atome peut prendre deux valeurs de vérité différentes.

Un type possible pour représenter une valuation est la **liste associative**

```
type valuations_assoc_list = (char * valeurs_verite) list
```

Exemple

La valuation précédente est représentée par la liste associative

```
[('p', Vrai); ('q', Faux); ('r', Vrai)]
```

Une valuation se représente élégamment par une **fonction** attribuant à un atome sa valeur de vérité

```
type valuations = char -> valeurs_verite
```

Exemple

La valuation précédente est représentée par la fonction ci-contre.

Celle-ci n'est cependant pas correcte car le filtrage de motifs y serait incomplet.

```
let valu a =
    match a with
        |'p' -> Vrai
        |'q' -> Faux
        |'r' -> Vrai
```

Ce filtrage se complète en **levant une exception** dans les cas problématiques via `raise`, fonction de type `exn -> 'a`, où `exn` est le type des exceptions. Une exception `EX` se définit via `exception EX`.

Exemple

La valuation précédente devient correcte ainsi.

Une clause joker est ajoutée. Celle-ci lève l'exception `Erreur` définie en premier lieu.

Notons la factorisation des cas similaires pour `'p'` et `'r'`.

```
exception Erreur
let valu a =
    match a with
        |'p' |'r' -> Vrai
        |'q' -> Faux
        |_ -> raise Erreur
```

Nous pouvons enfin mettre en place la fonction d'évaluation.

Elle s'écrit très simplement au moyen d'un filtrage :

```
let rec evaluer valu form =
    match form with
        |Atome a -> valu a
        |Non f -> if evaluer valu f = Faux then Vrai else Faux
        |Et (f1, f2) -> if evaluer valu f1 = Vrai && evaluer valu f2 = Vrai then Vrai else Faux
        |Ou (f1, f2) -> if evaluer valu f1 = Faux && evaluer valu f2 = Faux then Faux else Vrai
```

Exemple

En considérant la formule précédente $(p \vee q) \wedge (q \wedge (\neg r))$ liée au non `f` et sa valuation précédente

$$p \mapsto \text{vrai}, \quad q \mapsto \text{faux}, \quad r \mapsto \text{vrai},$$

liée au nom `valu`, nous l'évaluons par

```
match evaluer valu f with
    |Vrai -> print_endline "Vrai"
    |Faux -> print_endline "Faux"
```

Ceci imprime `Faux`.

12.3. Type option

Une *valeur optionnelle* de type `T` est

- soit une **valeur spéciale** ;
- soit une valeur de type `T` **encapsulée**.

Elles permettent d'écrire des fonctions qui, en cas d'échec, renvoient la valeur spéciale.

Ceci est implanté dans la bibliothèque standard par le type paramétré `'a option`, dont la définition est

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

Exemple

Pour éviter les erreurs de division par `0`, nous mettons en place la **division sûre** par

```
# let division_sure a b =
    if b = 0 then
        None
    else
        Some (a / b)
val division_sure : int -> int -> int option = <fun>
```

```
# division_sure 32 0;;
- : 'a option = None

# division_sure 32 4;;
- : int option = Some 8
```

Les fonctions à type de retour optionnel offrent une **alternative fonctionnelle au mécanisme des exceptions**, qui ont le désavantage de ne pas apparaître dans les types des fonctions.

La fonction `Option.get` de type `'a option -> 'a` permet de renvoyer la valeur encapsulée par une valeur optionnelle. Une exception est levée lorsque cette fonction est appliquée à `None`.

Exemples

```
# Option.get (division_sure 32 4);;
- : int = 8
```

```
# Option.get (division_sure 32 0);;
Exception: Invalid_argument "option is None".
```

Les fonctions `Option.is_some` et `Option.is_none` de type `'a option -> bool` testent respectivement si une valeur optionnelle encapsule une valeur ou si il s'agit de la valeur spéciale.

Exemples

```
# Option.is_some (Some "abc");;
- : bool = true
```

```
# Option.is_none (Some "abc");;
- : bool = false
```

La fonction `Option.map` de type `('a -> 'b) -> 'a option -> 'b option` est telle que `Option.map f opt` renvoie `Some (f v)` si `opt` encapsule la valeur `v` et `None` sinon.

Exemples

```
# Option.map succ (Some 1);;
- : int option = Some 2
```

```
# Option.map succ None;;
- : int option = None
```

La fonction `Option.bind` de type `'a option -> ('a -> 'b option) -> 'b option` est telle que `Option.bind opt f` renvoie `f v` si `opt` encapsule la valeur `v` et `None` sinon.

Exemples

```
# Option.bind (Some 1) (fun n -> Some (succ n));;
- : int option = Some 2
```

```
# Option.bind None (fun n -> Some (succ n));;
- : int option = None
```

Revenons à la fonction `Option.get`. Elle possède comme défaut majeur de lever une exception lorsqu'elle est appliquée sur `None`.

Une fonction importante à considérer est

```
# let get_default opt def =
      match opt with
          |None -> def
          |Some v -> v
val get_default : 'a option -> 'a -> 'a = <fun>
```

Elle est telle que `get_default opt def` renvoie la valeur encapsulée par `opt` si elle existe et renvoie `def` sinon.

Exemple

Dans le contexte de la représentation des formules logiques vu précédemment, une valuation peut se représenter par une valeur de vérité encapsulée. Ainsi,

```
exception Erreur
let valu a =
    match a with
        |'p' |'r' -> Vrai
        |'q' -> Faux
        |_ -> raise Erreur
```

devient

```
let valu a =
    match a with
        |'p' |'r' -> Some Vrai
        |'q' -> Some Faux
        |_ -> None
```

L'ancien code se met facilement à jour en utilisant les fonctions précédentes du module `Option`.

13. Fonctions d'ordre supérieur

13.1. Principes généraux

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 peut en **créer à la volée** et en appeler.

Rappel : étant donné que les fonctions sont conceptualisées de manière curryfiée, une fonction à $n \geqslant 2$ entrées est une fonction à une entrée qui **renvoie une fonction** à $n-1$ entrées. Ainsi, toute fonction à $n \geqslant 2$ entrées est elle-même déjà une fonction d'ordre supérieur.

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))
```

Nous inférons le type

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

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'est-à-dire,

$$f^n(x)$$

Par exemple

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

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

Exemple

Analysons le type de la fonction

```
let compose f1 f2 =
    fun x -> f1 (f2 x)
```

Nous inférons le type

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

qui montre que `compose` est paramétrée par deux fonctions `f1` et `f1`, de types respectifs `'a -> 'b` et `'c -> 'a`. C'est donc une fonction d'ordre supérieur.

Elle calcule la fonction définie comme la composée de `f1` avec `f2`, c'est-à-dire,

$$f1 \circ f2$$

Par exemple,

```
# let f = compose succ succ;;
val f : int -> int = <fun>

# f 0;;
- : int = 2
```

```
# compose int_of_float Float.round 2.5001;;
- : int = 3
```

L'opérateur d'*application inversée* est l'opérateur binaire `|>`. Il est de type

```
'a -> ('a -> 'b) -> 'b
```

Il permet, étant données une fonction `f` de type `'a -> 'b` et une expression `e` de type `'a`, d'écrire

```
e |> f
```

à la place de

```
f e
```

Il sert à rendre les **applications successives de fonctions plus naturelles**.

En effet, si `e` est une expression de type `'a0` et chaque f_i est une fonction de type $'a_{i-1}$ `->` $'a_i$, il devient possible d'écrire

```
e |> f1 |> f2 |> ... |> fn
```

à la place de

```
fn (... (f2 (f1 e)) ...)
```

Remarque : pour bien s'articuler avec cet opérateur, il est intéressant d'écrire les fonctions avec le paramètre principal en dernière position.

13.2. Mots fonctionnels

Considérons la façon fonctionnelle de représenter les mots vue précédemment via le type

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

Rappelons 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
}
```

Ceci représente représente le mot `abb` dont les lettres de type `char`.

Remarque : il est encore meilleur de faire en sorte que le champ `lettres` soit de type `int -> 'a option` afin de renvoyer `None` dans le cas où on essaye de lire une lettre à une position non autorisée. Ceci n'est pas illustré ici.

La fonction d'ordre supérieur ci-contre renvoie la chaîne de caractères qui représente le mot `u` :

```
let vers_chaine lettre_vers_char u =
    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 d'ordre supérieur est de type `('a -> char) -> 'a mot -> string` ;
- `lettre_vers_char` est une fonction qui renvoie le caractère associé à toute lettre de type `'a` ;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` faite de caractères `c`.

Exemples

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

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

La fonction ci-contre renvoie le mot défini comme étant la concaténation des deux mots `u` et `v` :

```
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 :

- cette fonction est de type `'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`. Elle se base 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
  vers_chaine (fun x -> x) mot_4;;
- : string = "abbabb"
```

Nous pouvons 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))
```

Exemples

Avec la liaison préalable `let vers_chaine_char = vers_chaine (fun x -> x)` qui, au passage, est une application partielle,

```
# let w = mot_fibo 0 in vers_chaine_char w;;
- : string = "b"

# let w = mot_fibo 1 in vers_chaine_char w;;
- : string = "a"

# let w = mot_fibo 2 in vers_chaine_char w;;
- : string = "ab"
```

```
# let w = mot_fibo 3 in vers_chaine_char w;;
- : string = "aba"

# let w = mot_fibo 4 in vers_chaine_char w;;
- : string = "abaab"

# let w = mot_fibo 5 in vers_chaine_char w;;
- : string = "abaababa"
```

13.3. Images fonctionnelles

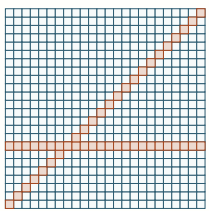
Considérons la façon fonctionnelle de représenter les images vue précédemment via les types ci-contre :

```
type points = int * int
type 'a images = {
    contenus_pixels : points -> 'a;
    largeur : int;
    hauteur : int
}
type couleurs = {rouge : int; vert : int; bleu : int}
```

Exemple

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y || y = 7 then
            {rouge = 127; vert = 127; bleu = 127}
        else
            {rouge = 255; vert = 255; bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

Le nom `im_3` est lié à la valeur de type `couleurs images` qui représente l'image (dans le plan cartésien positif)



Remarque : ici aussi, le type `points -> 'a option` pour le champ `contenus_pixels` est une excellente alternative.

La fonction suivante permet d'ouvrir une fenêtre graphique et de dessiner l'image `im`.

```
let dessiner im =
    Graphics.open_graph (Printf.sprintf " %dx%d" im.largeur im.hauteur);
    List.init im.largeur
        (fun x ->
            List.init im.hauteur
                (fun y ->
                    let c = im.contenus_pixels (x, y) in
                    let c' = Graphics.rgb c.rouge c.vert c.bleu in
                    (x, y, c')))
    |> List.flatten
    |> List.iter (fun (x, y, c) -> Graphics.set_color c; Graphics.plot x y);
    Graphics.wait_next_event [Button_down; Key_pressed] |> ignore;
    Graphics.close_graph ()
```

Quelques explications :

- cette fonction utilise le module `Graphics` permettant d'ouvrir des fenêtres graphiques, de dessiner à l'intérieur et de gérer les entrées sur clavier et souris ;
- la liste des triplets contenant les coordonnées des pixels et leur couleur est construite. Ensuite, cette liste est parcourue pour afficher chaque pixel ;
- une fois l'image dessinée, la fenêtre reste ouverte jusqu'à ce que l'utilisateur clique ou appuie sur une touche.

Voici des fonctions pour calculer l'image obtenue en complémentant les couleurs des pixels d'une image `im` :

```
let complementer_couleur c =
    let cpl x = 255 - x in
    {rouge = cpl c.rouge; vert = cpl c.vert; bleu = cpl c.bleu}

let complementer im =
    let contenus_pixels p =
        complementer_couleur (im.contenus_pixels p)
    in
    {im with contenus_pixels = contenus_pixels}
```

Tout appel `complementer im` s'évalue en temps $\Theta(1)$. Les complexités en temps et en espace ne dépendent pas de la taille de l'image.

En revanche, pour effectivement dessiner l'image, il n'est pas possible de faire autrement que de considérer tous les pixels et de les afficher un par un. Dans ce cas, la complexité en temps pour afficher le complémentaire d'une image est $\Theta(nm)$ où n est sa largeur et m sa hauteur.

Voici une fonction qui calcule l'**image miroir** d'une image `im` :

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

Voici des fonctions pour **superposer** deux images `im1` et `im2` de mêmes dimensions en considérant les moyennes de composantes de couleurs entre chaque pixel en mêmes positions :

```
let moyenne_couleurs c1 c2 =
    let moy x y = (x + y) / 2 in
    {rouge = moy c1.rouge c2.rouge; vert = moy c1.vert c2.vert; bleu = moy c1.bleu c2.bleu}

let superposer im1 im2 =
    let contenus_pixels p =
        moyenne_couleurs (im1.contenus_pixels p) (im2.contenus_pixels p)
    in
    {im1 with contenus_pixels = contenus_pixels}
```

Les mêmes remarques précédentes pour les complexités de ces deux fonctions s'appliquent.

Exemple

Voici un exemple d'image qu'il est possible de construire en utilisant les transformations précédentes :

```
let im_alien =
    let noir = {rouge = 0; vert = 0; bleu = 0} in
    let blanc = {rouge = 255; vert = 255; bleu = 255} in
    let cercle x0 y0 r p =
        let (x, y) = p in
        let dx = x - x0 and dy = y - y0 in
        if dx * dx + dy * dy <= r * r then noir else blanc
    in
    let im_oeil_gauche =
        {contenus_pixels = cercle 90 130 20; largeur = 240; hauteur = 240}
        |> complementer
    in
    let im_tete = {contenus_pixels = cercle 120 120 60; largeur = 240; hauteur = 240} in
    let im_yeux = superposer im_oeil_gauche (miroir im_oeil_gauche) in
    superposer im_yeux im_tete
```