Programmation fonctionnelle

Fiche de TP 7

L3 Informatique 2020-2021

Images fonctionnelles

Il existe plusieurs façons de représenter des images numériques. En programmation fonctionnelle, un moyen élégant pour cela consiste à coder une image par une fonction qui associe à tout point du plan une couleur, comme vu en cours[1].

Dans ce TP, nous allons utiliser cette idée pour pour dessiner des images à l'aide du module `Graphics` de CAML. On définit le type suivant pour les images :

```
type picture = int * int -> Graphics.color
```

Rappelons que connaître une image, c'est savoir pour chaque pixel de celle-ci la couleur qu'il porte. Il existe ainsi au moins deux manières de procéder pour coder une image. La première, de **nature impérative**, consiste à coder une image par un tableau à deux dimensions de couleurs. La deuxième, de **nature fonctionnelle**, consiste à représenter une image par une fonction paramétrée par un pixel (des coordonnées) et qui renvoie la couleur qu'il porte. C'est de cette façon que nous allons nous y prendre.

Il est fourni un fichier `funpics_skeleton.ml` dans lequel apparaît une fonction `render` qui permet d'afficher une image de dimensions $w \times h$ pour les valeurs de w et h définies dans ce même fichier.

Voici à présent deux exemples de définitions d'images colorées que l'on peut afficher immédiatement à l'aide de la fonction `render` :

```
let black_on_black : picture = fun point -> black
let half_plane color : picture =
  fun (x, y) -> if x < w / 2 then color else background
```

Afficher ces images dans des fenêtres graphiques de tailles 512×512 et 768×512. Pour la deuxième image, on fera en sorte d'afficher le demi-plan en rouge (couleur `red`).

La figure 1 montre quelques images que nous définirons dans ce TP.

1. Nous adoptons ici des conventions légèrement différentes de celles du cours.

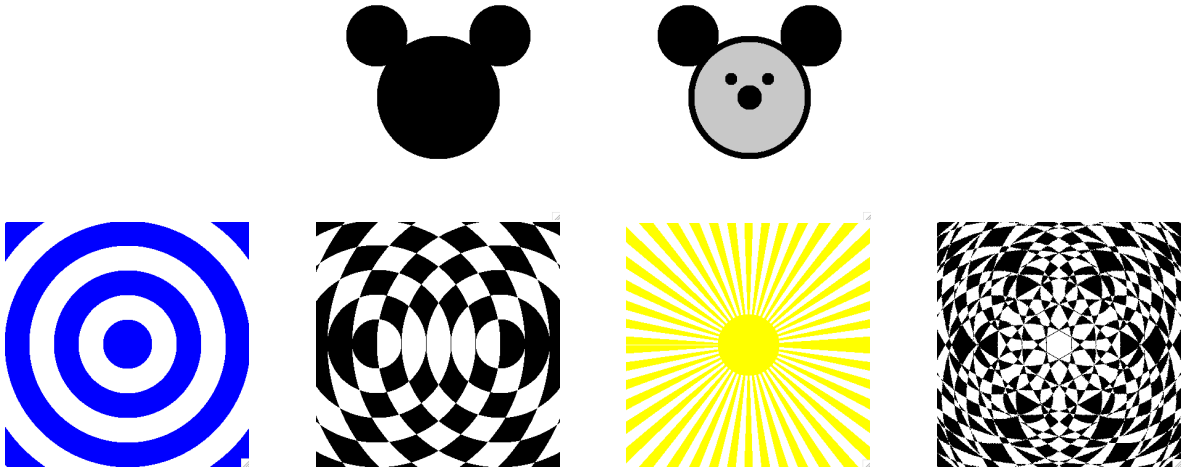

FIGURE 1 – Quelques exemples d'images : l'ombre de Mickey, Mickey, des cercles concentriques bleus, des cercles concentriques doubles noirs, des rayons de soleil et une rosace.

Exercice 1. (Images et transformations simples)

Dans cet exercice, nous allons dessiner des formes géométriques simples et leur appliquer des transformations pour les déplacer.

1. Écrire une fonction

```
val diagonal : Graphics.color -> picture = <fun>
```

qui, étant donnée une couleur en argument, renvoie une image qui représente la diagonale de la couleur spécifiée (de largeur 1 pixel) et qui démarre en bas à gauche de l'image.

2. Écrire une fonction

```
val square : int -> Graphics.color -> picture = <fun>
```

qui, étant donné une couleur et la longueur du coté (en nombre de pixels) en arguments, renvoie une image qui représente un carré de la couleur spécifiée, de la taille spécifiée et centré en $(0, 0)$.

3. Écrire une fonction

```
val rectangle : int -> int -> Graphics.color -> picture = <fun>
```

qui, étant donné une couleur, une largeur et une hauteur en arguments, renvoie une image qui représente un rectangle selon les caractéristiques spécifiées et centré en $(0, 0)$.

4. Écrire une fonction

```
val disk : int -> Graphics.color -> picture = <fun>
```

qui, étant donné une couleur et un rayon en arguments, renvoie une image qui représente un disque selon les caractéristiques spécifiées et centré en $(0, 0)$.

5. Écrire une fonction

```
val circle : int -> Graphics.color -> picture = <fun>
```

qui, étant donné une couleur et un rayon en arguments, renvoie une image qui représente un cercle selon les caractéristiques spécifiées et centré en $(0, 0)$.

6. On souhaite pouvoir déplacer une image. Déplacer une image consiste appliquer une translation sur chacun de ses pixels. Écrire une fonction

```
val move : picture -> int * int -> picture = <fun>
```

qui, étant donnée une image et un point en arguments, renvoie l'image obtenue en centrant l'entrée sur le point. Autrement dit, si `im` est une image, `move im (x, y)` est l'image dans laquelle le pixel de coordonnées $(0, 0)$ est celui de coordonnées `(x, y)` de `im`, le pixel de coordonnées $(1, 0)$ est celui de coordonnées `(x + 1, y)` de `im`, *etc.*

7. Créer une image contenant un rectangle vert de taille 128×64 centré en son coin haut gauche.

8. Écrire une fonction

```
val vertical_symmetry : picture -> picture = <fun>
```

qui, étant donnée une image, renvoie l'image obtenue par symétrie selon la droite verticale qui passe par $(0, 0)$.

9. Écrire une fonction

```
val horizontal_symmetry : picture -> picture = <fun>
```

qui, étant donnée une image, renvoie l'image obtenue par symétrie selon la droite horizontale qui passe par $(0, 0)$.

10. Créer une image contenant une antidiagonale verte en utilisant les fonctions `diagonal` et `vertical_symmetry`.

Exercice 2. (Où l'on complique un tout petit peu les choses)

On souhaite pouvoir dessiner des images un peu plus sophistiquées et notamment des motifs qui se répètent.

1. Écrire une fonction

```
val v_lines : int -> picture = <fun>
```

qui, étant donné un entier n en argument, renvoie une image qui représente des rayures noires verticales fines (de largeur 1 pixel) espacées de n pixels.

2. Écrire une fonction

```
val v_stripes : int -> picture = <fun>
```

qui, étant donné un entier n en argument, renvoie une image qui représente des rayures noires verticales larges (de largeur n pixels) espacées de n pixels.

3. Écrire une fonction

```
val chessboard : Graphics.color -> int -> picture = <fun>
```

qui, étant donnés une couleur et un entier n en arguments, renvoie une image qui représente un échiquier dont les cases font n pixels de côté.

4. Écrire une fonction

```
val concentric : Graphics.color -> int -> picture = <fun>
```

qui, étant donnés une couleur et un entier n en arguments, renvoie une image qui représente des anneaux concentriques (blanc et colorés alternés) de largeur n (voir la figure 1).

Exercice 3. (Composition)

Dans cet exercice, on va composer des images entre elles.

1. Écrire une fonction

```
val compose_two : picture -> picture -> picture = <fun>
```

qui, étant données deux images en arguments, renvoie une nouvelle image qui est la superposition des deux (la seconde est « au dessus » de la première).

Plus exactement, la composition fonctionne pixel par pixel de la manière suivante. Tout pixel p du résultat est de la couleur spécifiée par la table

p dans 1re im. \ p dans 2re im.	`background`	c_2
`background`	`background`	c_2
c_1	c_1	c_2

où c_1 (resp. c_2) est la couleur de p dans la 1re (resp. 2e) image quand différente de la couleur de fond `background`.

2. Utiliser la composition pour dessiner une ombre de Mickey (voir la figure 1). Utiliser pour cela aussi la fonction `move`.

3. Pour pouvoir superposer facilement autant d'images que l'on souhaite, écrire une fonction

```
val compose : picture list -> picture = <fun>
```

qui, étant donné une liste d'images en argument, renvoie l'image obtenue en les superposant les unes sur les aux autres, avec la dernière image de la liste « au dessus ».

4. Utiliser la composition pour dessiner une tête de Mickey (voir la figure 1).

Exercice 4. (Ça tourne)

On souhaite maintenant réaliser des rotations sur les images. Pour cela, le plus simple est d'utiliser des coordonnées polaires au lieu des coordonnées cartésiennes. Un point (ρ, θ) est défini par sa distance ρ à $(0, 0)$ et un angle θ par rapport à l'axe des abscisses.

Deux fonctions de conversion entre coordonnées polaires et cartésiennes sont fournies dans le fichier `funpics_skeleton.ml`

1. Écrire une fonction

```
val rotate : picture -> float -> picture = <fun>
```

qui, étant donné une image et un angle (en radians) en arguments, renvoie la rotation de cette image suivant l'angle spécifié.

2. Créer l'image obtenue en tournant Mickey d'un quart de tour dans le sens direct.
3. Écrire une fonction

```
val sun : Graphics.color -> picture = <fun>
```

qui, étant donné une couleur en argument, renvoie l'image représentant un soleil de la couleur spécifiée (voir la figure 1).

4. Écrire une fonction

```
val compose_xor_two : picture -> picture -> picture = <fun>
```

qui, étant donné deux images en arguments, renvoie l'image définie comme étant la superposition/annulation des deux.

Plus exactement, la superposition/annulation fonctionne pixel par pixel de la manière suivante. Tout pixel p du résultat est de la couleur spécifiée par la table

p dans 1re im. \ p dans 2re im.	`background`	c_2
`background`	`background`	c_2
c_1	c_1	`background`

où c_1 (resp. c_2) est la couleur de p dans la 1re (resp. 2e) image quand différente de la couleur de fond `background` (voir l'image des cercles concentriques doubles de la figure 1).

5. Écrire une fonction

```
val double_concentric : int -> picture = <fun>
```

qui, étant donnée une largeur w en argument, renvoie une image en noir et blanc qui représente des anneaux concentriques doubles de largeur w (voir la figure 1).

6. Écrire une fonction

```
val compose_xor : picture list -> picture = <fun>
```

qui, étant donné une liste d'images en argument, renvoie l'image obtenue en les superposant/annulant.

7. Écrire une fonction

```
val rosace : picture = <fun>
```

qui renvoie une image en noir et blanc qui représente une rosace (par rotation d'anneaux concentriques imbriqués, voir la figure 1).

Exercice 5. *(Bonus)*

1. Écrire une fonction

```
val sierpinski : picture = <fun>
```

qui renvoie l'image en noir et blanc dont tous les pixels (x, y) tels que `x lor y = x` sont noirs (l'opérateur `lor` est un « ou » bit à bit).

2. Écrire une fonction

```
val mandelbrot : picture = <fun>
```

qui renvoie une image en noir et blanc offrant une représentation de la fractale de Mandelbrot.