Génération d'images en Haskell

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

Fabian Reiter
fabian.reiter@univ-eiffel.fr

30 novembre 2021

1 Travail demandé

Le travail est à faire en binôme.

Le projet est à rendre sous forme d'un dépôt GitHub privé [1] au plus tard le 19 décembre 2021. L'utilisation de `stack` est impérative. Votre projet devra pouvoir être complilé par `stack build` après clonage du dépôt et l'accès à la bibliothèque par `stack exec - ghci`.

Une attention particulière sera portée à la clareté du code. En particulier, privilégiez la composition et la réutilisation de fonctions simples. « *Premature optimization is the root of all evil* ». D. Knuth.

N'oubliez pas de documenter - succintement - vos modules et fonctions. La génération de la documentation est déclenchée par `stack haddock` (`https://www.haskell.org/haddock/doc/html/markup.html`). Des tests unitaires ne sont pas demandés mais seront favorablement considérés ☺ (`stack test`). La section 8 (page 12) propose quelques pistes pour aller plus loin si vous avez terminé.

Par ailleurs, pensez à **préciser vos noms** complets dans un fichier `README.md` à la racine du projet. (C'est important pour que nous puissions vous attribuer une note.)

2 Introduction

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 valeur.

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, et nous n'utiliserons que deux couleurs : le blanc et le noir.

L'approche fonctionnelle permet donc de distinguer la géométrie de l'image, et plus généralement, le modèle de la présentation. Pour illustrer ce concept, votre travail se portera uniquement ici sur le modèle.

1. Il faudra évidemment nous donner le droit d'y accéder. Nos identifiants GitHub sont `vialette` et `fabian-reiter`.

3 Les modules

Notre projet consistera en un module `Main` [donné, pas nécessaire de le comprendre], et une bibliothèque `Data.FImage`. Cette dernière sera composée au final au moins des modules suivants :

- `Data.FImage.Algebra` pour les opérations algébriques sur les images **[à écrire]**,
- `Data.FImage.BMP` [donné, pas nécessaire de le comprendre],
- `Data.FImage.Geometry` pour quelques fonctions simples calculant la distance entre deux points et convertissant des coordonnées cartésiennes en coordonnées polaires, ou vice versa **[à compléter]**,
  - `Data.FImage.Geometry.Point` pour la définition du type `Point` [donné, à comprendre],
  - `Data.FImage.Geometry.PolarPoint` pour la définition du type `PolarPoint` [donné, à comprendre],
  - `Data.FImage.Geometry.Vector` pour la définition du type `Vector` [donné, à comprendre],
- `Data.FImage.Image` pour la définition du type `Image` [donné, à comprendre],
  - `Data.FImage.Image.Gallery` pour une collection d'images construites à l'aide de transformations géométriques et d'opérations algébriques sur les images [donné, à comprendre],
  - `Data.FImage.Image.Simple` pour une collection d'images basiques [donné, à comprendre],
- `Data.FImage.Interval` [donné, pas nécessaire de le comprendre],
- `Data.FImage.Render` [donné, pas nécessaire de le comprendre],
- `Data.FImage.Transformation` pour les transformations géométriques sur les images **[à compléter]**,
- `Data.FImage.View` [donné, pas nécessaire de le comprendre],
- `Data.FImage.Window` [donné, pas nécessaire de le comprendre],

Vous pouvez bien sûr ajouter d'autres modules à votre projet. Les modules `Data.FImage.BMP`, `Data.FImage.Interval`, `Data.FImage.Render`, `Data.FImage.View`, et `Data.FImage.Window` sont utilisés pour les entrées-sorties (en particulier, pour enregistrer des extraits d'images dans des fichiers). Cette thématique allant au-delà des notions abordées en cours, il n'est pas nécessaire de comprendre ces modules en détail. (Cependant, si cela vous intéresse et que vous voulez aller un peu plus loin, le chapitre 9 du livre *"Learn You a Haskell for Great Good!"* peut être un bon point de départ.)

Le module `Main`, quand à lui, utilise les fonctions définies dans `Data.FImage.BMP` pour créer des fichiers BMP à partir des images définies dans le projet. Là encore, il n'est pas nécessaire de comprendre tous les détails. (Cependant, si vous souhaitez définir vos propres images, il faudra les ajouter à la liste des images dans la fonction `main` pour que les fichiers BMP correspondants soient générés.)

4 Les types Haskell

Pour les images, nous utiliserons obligatoirement le type suivant :

```
type Image = Point -> Bool
```

où le type `Point` est défini comme une paire de coordonnées cartésiennes :

```
data Point = Point {x :: Float, y :: Float} deriving (Show, Eq, Ord)
```

Nous considérons donc une image comme une fonction qui attribue une couleur (soit blanc, soit noir) à chaque point du le plan (infini). Ceci est complètement équivalent à dire qu'une image *est* une surface infinie dont chaque point est coloré soit en blanc soit en noir.

Dans certains cas, il est plus pratique d'utiliser des coordonnées polaires que des coordonnées cartésiennes. On peut alors se servir du type `PolarPoint` défini par :

```
type Angle = Float
data PolarPoint = PolarPoint {rho :: Float, theta :: Angle} deriving (Show, Eq, Ord)
```

Afin que vous puissiez vous servir de ce type auxiliaire, on vous demandera (cf. section 5) d'écrire deux fonctions de conversion pour passer facilement d'une représentation à une autre : `toPolar :: Point -> PolarPoint` et `fromPolar :: PolarPoint -> Point`.

Pour les transformations d'images, nous utiliserons obligatoirement le type suivant :

```
type Transformation = Image -> Image
```

Une transformation est donc simplement une fonction qui convertit une image donnée en une nouvelle image. Il est utile ici de se rappeler qu'une image est également une fonction (`Point -> Bool`), et que nous avons donc

```
type Transformation = Point -> Bool -> Point -> Bool
```

Certaines transformations géométriques (translation, agrandissement, et réduction) seront paramétrées par un vecteur. Pour cela, nous utiliserons le type :

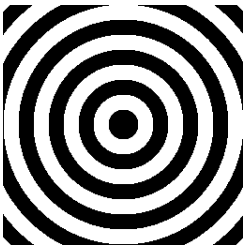
```
data Vector = Vector {x:: Float, y :: Float} deriving (Show, Eq, Ord)
```

Vous remarquerez que la définition de ce type est identique à celle de `Point`. Il est néanmoins bon d'avoir deux types distincts car nous en ferons des usages différents, et cela permet au compilateur de détecter certaines erreurs de type. (Le compilateur est votre ami. Plus il détecte d'erreurs, moins il en restera lors de l'exécution du programme.)

Les modules suivants sont donnés complets :
— `Data.FImage.Image` en section 9 (page 13),
— `Data.FImage.Image.Gallery` en section 10 (page 14).
— `Data.FImage.Image.Simple` en section 11 (page 17).
— `Data.FImage.Geometry.Point` en section 12 (page 20),
— `Data.FImage.Geometry.PolarPoint` en section 13 (page 21), et
— `Data.FImage.Geometry.Vector` en section 14 (page 22).
Les modules suivants sont donnés incomplets :
— `Data.FImage.Geometry` en section 15 (page 24).
— `Data.FImage.Transformation` en section 16 (page 25).

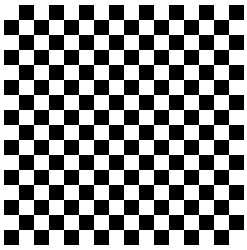
5 Coordonnées polaires

Écrire les fonctions (module `Data.FImage.Geometry`) :

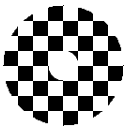


altRings

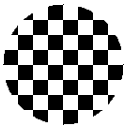
annulus.bmp



checker

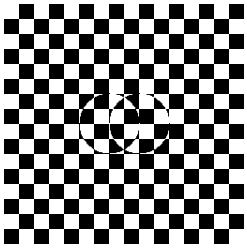


checkerAnnulus.bmp

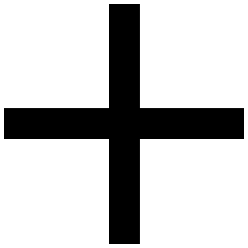


checkerDisk

checkerSquare



checkerXORCircles2



cross.bmp

diamond1

diamond2

disk

eclipse.bmp

hHalfPlane

hHalfPlane0

hStrip

hTranslateUSquare.bmp

FIGURE 1 – Images crées par `Main.hs`.

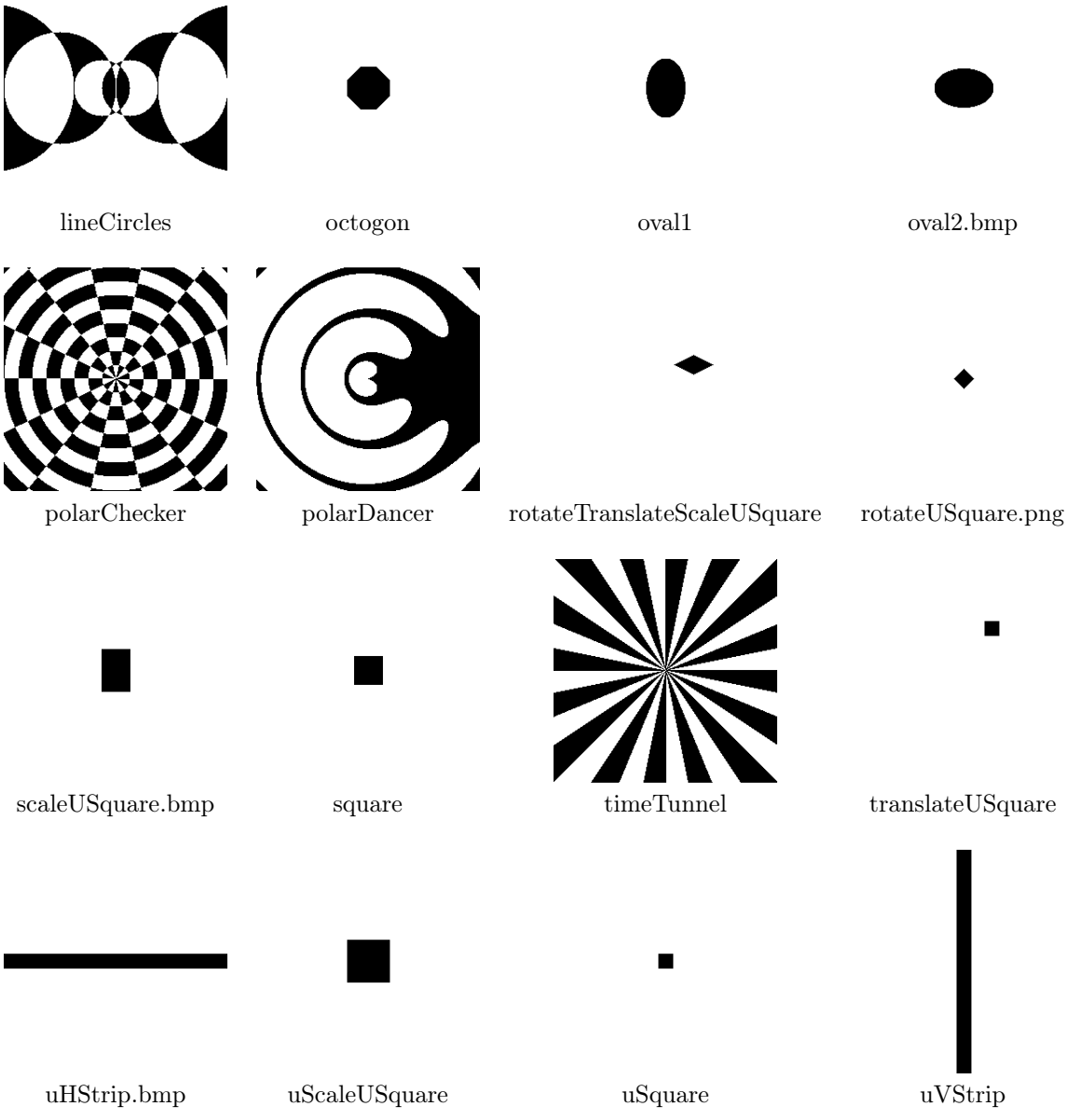
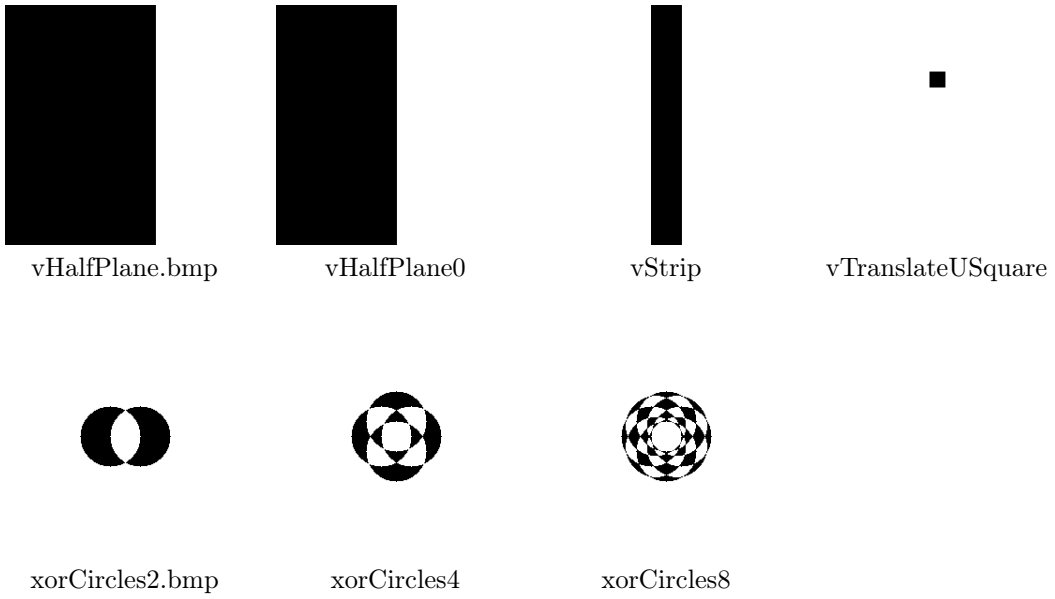


FIGURE 2 – Images crées par `Main.hs`



vHalfPlane.bmp vHalfPlane0 vStrip vTranslateUSquare

xorCircles2.bmp xorCircles4 xorCircles8

FIGURE 3 – Images crées par `Main.hs`

```haskell
-- | Convert a polar point to a cartesian point.
fromPolar :: PolarPoint.PolarPoint -> Point.Point

-- | Convert a cartesian point to a polar point.
toPolar :: Point.Point -> PolarPoint.PolarPoint
```

Vous pouvez tester vos fonctions avec les images `Data.FImage.Image.Simple.polarChecker` et `Data.FImage.Image.Simple.polarDancer` :

```haskell
-- | Polar checker. The parameter determines the number of alternations, and
-- hence is twice the number of slices.
-- For a cartesian point (x, y), convert to polar point (t', r') and convert
-- back to a cartesian point (r, t * n + pi) and use checker function.
polarChecker :: Float -> Image.Image
polarChecker f = checker . aux . Geometry.toPolar
  where
    aux p = Point.mk x y
      where
        x = PolarPoint.rho p
        y = f / pi * PolarPoint.theta p

-- | Polar dancer. The parameter determines the "thickness of the dancer's arms".
polarDancer :: Float -> Image.Image
polarDancer f = vStrip f . Geometry.fromPolar . aux . Geometry.toPolar
```

```
  where
    aux p = PolarPoint.mk rho' theta'
      where
        rho' = PolarPoint.theta p
        theta' = PolarPoint.rho p
```

6 Transformations géométriques

Écrire les fonctions du module `Data.FImage.Transformation` :

```
module Data.FImage.Transformation
(
  -- * type
  Transformation
-- * Transforming boolean images
, translate
, hTranslate
, vTranslate
, scale
, hScale
, vScale
, uScale
, rotate
)
where

import qualified Data.FImage.Image           as Image
import qualified Data.FImage.Geometry.Point  as Point
import qualified Data.FImage.Geometry.Vector as Vector

type Transformation = Image -> Image

{- | Translate a Boolean image according to a vector.

>>> import qualified Data.FImage.Geometry.Vector as Vector
>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = Point.x p > 0 && Point.y p > 0
>>> v = Vector.mk 1 2
>>> mapM_ print [let p = Point.mk x y in (p, translate v f p) | x <- [0 .. 3], y <- [0 .. 3]]
(Point {x = 0.0, y = 0.0},False)
(Point {x = 0.0, y = 1.0},False)
(Point {x = 0.0, y = 2.0},False)
(Point {x = 0.0, y = 3.0},False)
(Point {x = 1.0, y = 0.0},False)
(Point {x = 1.0, y = 1.0},False)
(Point {x = 1.0, y = 2.0},False)
```

```
(Point {x = 1.0, y = 3.0},False)
(Point {x = 2.0, y = 0.0},False)
(Point {x = 2.0, y = 1.0},False)
(Point {x = 2.0, y = 2.0},False)
(Point {x = 2.0, y = 3.0},True)
(Point {x = 3.0, y = 0.0},False)
(Point {x = 3.0, y = 1.0},False)
(Point {x = 3.0, y = 2.0},False)
(Point {x = 3.0, y = 3.0},True)
-}
translate :: Vector.Vector -> Transformation

{- | Translate a Boolean image horizontally by a given distance.

>>> import qualified Data.FImage.Geometry.Vector as Vector
>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = Point.x p > 0 && Point.y p > 0
>>> mapM_ print [let p = Point.mk x y in (p, hTranslate 1 f p) | x <- [0 .. 3], y <- [0 .. 3]]
(Point {x = 0.0, y = 0.0},False)
(Point {x = 0.0, y = 1.0},False)
(Point {x = 0.0, y = 2.0},False)
(Point {x = 0.0, y = 3.0},False)
(Point {x = 1.0, y = 0.0},False)
(Point {x = 1.0, y = 1.0},False)
(Point {x = 1.0, y = 2.0},False)
(Point {x = 1.0, y = 3.0},False)
(Point {x = 2.0, y = 0.0},False)
(Point {x = 2.0, y = 1.0},True)
(Point {x = 2.0, y = 2.0},True)
(Point {x = 2.0, y = 3.0},True)
(Point {x = 3.0, y = 0.0},False)
(Point {x = 3.0, y = 1.0},True)
(Point {x = 3.0, y = 2.0},True)
(Point {x = 3.0, y = 3.0},True)
-}
hTranslate :: Float -> Transformation

{- | Translate a Boolean image vertically by a given distance.

>>> import qualified Data.FImage.Geometry.Vector as Vector
>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = Point.x p > 0 && Point.y p > 0
>>> mapM_ print [let p = Point.mk x y in (p, vTranslate 2 f p) | x <- [0 .. 3], y <- [0 .. 3]]
(Point {x = 0.0, y = 0.0},False)
(Point {x = 0.0, y = 1.0},False)
(Point {x = 0.0, y = 2.0},False)
```

```
(Point {x = 0.0, y = 3.0},False)
(Point {x = 1.0, y = 0.0},False)
(Point {x = 1.0, y = 1.0},False)
(Point {x = 1.0, y = 2.0},False)
(Point {x = 1.0, y = 3.0},True)
(Point {x = 2.0, y = 0.0},False)
(Point {x = 2.0, y = 1.0},False)
(Point {x = 2.0, y = 2.0},False)
(Point {x = 2.0, y = 3.0},True)
(Point {x = 3.0, y = 0.0},False)
(Point {x = 3.0, y = 1.0},False)
(Point {x = 3.0, y = 2.0},False)
(Point {x = 3.0, y = 3.0},True)
-}
vTranslate :: Float -> Transformation

{- | Scale a Boolean image according to a vector.

>>> import qualified Data.FImage.Geometry.Vector as Vector
>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = Point.x p >= 0 && Point.x <= 1 && Point.y >= 0 && Point.y <= 1
>>> v = Vector.mk 1 2
>>> mapM_ print [let p = Point.mk x y in (p, scale v f p) | x <- [0 .. 3], y <- [0 .. 3]]
(Point {x = 0.0, y = 0.0},True)
(Point {x = 0.0, y = 1.0},True)
(Point {x = 0.0, y = 2.0},True)
(Point {x = 0.0, y = 3.0},False)
(Point {x = 1.0, y = 0.0},True)
(Point {x = 1.0, y = 1.0},True)
(Point {x = 1.0, y = 2.0},True)
(Point {x = 1.0, y = 3.0},False)
(Point {x = 2.0, y = 0.0},False)
(Point {x = 2.0, y = 1.0},False)
(Point {x = 2.0, y = 2.0},False)
(Point {x = 2.0, y = 3.0},False)
(Point {x = 3.0, y = 0.0},False)
(Point {x = 3.0, y = 1.0},False)
(Point {x = 3.0, y = 2.0},False)
(Point {x = 3.0, y = 3.0},False)
-}
scale :: Vector.Vector -> Transformation

{- | Scale a Boolean image horizontally by a given factor. -}
hScale :: Float -> Transformation

{- | Scale a Boolean image vertically by a given factor. -}
```

```
vScale :: Float -> Transformation

{- | Scale a Boolean image uniformly by a given factor. -}
uScale :: Float -> Transformation

{- | Rotate a Boolean image uniformly by a given angle. -
rotate :: Float -> Transformation
```

7 Opérations algébriques

Écrire les fonctions du module `Data.FImage.Algebra`.

```
module Data.FImage.Algebra
(
  universe
, empty
, comp
, inter
, union
, xor
, diff
)
where

import qualified Data.FImage.Image as Image

{- | The all-true boolean image.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> [let p = Point.mk i j in universe p | i <- [0 .. 2], j <- [0 .. 2]]
[True,True,True,True,True,True,True,True,True]
-}
universe :: Image.Image

{- | The all-False boolean image.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> [let p = Point.mk i j in empty p | i <- [0 .. 2], j <- [0 .. 2]]
[False,False,False,False,False,False,False,False,False]
-}
empty :: Image.Image

{- | Complement a boolean image.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> let p = mk 0 0 in comp universe p
```

```
False
>>> let p = mk 0 0 in comp empty p
True
>>> f p = abs (Point.x p - Point.y p) >= 1
>>> [let p = Point.mk i j in comp f p | i <- [0 .. 2], j <- [0 .. 2]]
[True,False,False,False,True,False,False,False,True]
-}
comp :: Image.Image -> Image.Image

{- | Intersection of two boolean images.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = abs (Point.x p - Point.y p) >= 1
>>> g p = abs (Point.x p - Point.y p) <= 1
>>> [let p = Point.mk i j in inter f g p | i <- [0 .. 2], j <- [0 .. 2]]
[False,True,False,True,False,True,False,True,False]
-}
inter :: Image.Image -> Image.Image -> Image.Image

{- | Union of two boolean images.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = abs (Point.x p - Point.y p) >= 1
>>> g p = abs (Point.x p - Point.y p) <= 1
>>> [let p = Point.mk i j in union f g p | i <- [0 .. 2], j <- [0 .. 2]]
[True,True,True,True,True,True,True,True,True]
-}
union :: Image.Image -> Image.Image -> Image.Image

{- | Xor of two boolean images.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = abs (Point.x p - Point.y p) >= 1
>>> g p = abs (Point.x p - Point.y p) <= 1
>>> [let p = Point.mk i j in xor f g p | i <- [0 .. 2], j <- [0 .. 2]]
[True,False,True,False,True,False,True,False,True]
-}
xor :: Image.Image -> Image.Image -> Image.Image

{- | difference of two boolean images.

>>> import qualified Data.FImage.Geometry.Point as Point
>>> f p = abs (Point.x p - Point.y p) >= 1
>>> g p = abs (Point.x p - Point.y p) <= 1
>>> [let p = Point.mk i j in diff f g p | i <- [0 .. 2], j <- [0 .. 2]]
[False,False,True,False,False,False,True,False,False]
```

```
-}
diff :: Image.Image -> Image.Image -> Image.Image
```

8 Pour aller plus loin

Une première extension possible consiste à représenter des images en niveaux de gris. Pour cela nous pouvons introduire le type suivant :

```
type GImage = Point -> Float
```

Une image en niveaux de gris est donc une fonction qui attribue à chaque point un niveau de gris compris entre 0 et 1 (0 pour blanc et 1 pour noir).

Une seconde extension consiste à représenter des images en couleur. Nous pouvons introduire les types suivants :

```
data RGBColor = RGB {getRed :: Float, getGreen :: Float, getBlue :: Float}
type CImage = Point -> RGBColor
```

Une image en couleur est donc une fonction qui attribue à chaque point une couleur définie par ses composantes *rouge*, *verte* et *bleue*.

9 Data.FImage.Image

```haskell
module Data.FImage.Image
(
  -- * type
  Image
)
where

import qualified Data.FImage.Geometry.Point as Point

-- | Image (aka region) type definition
type Image = Point.Point -> Bool
```

10 Data.FImage.Image.Gallery

```
module Data.FImage.Image.Gallery
(
  diamond
, oval
, annulus
, checkerDisk
, checkerSquare
, checkerAnnulus
, octogon
, xorCircles2
, checkerXORCircles2
, xorCircles4
, xorCircles8
, eclipse
, lineCircles
, timeTunnel
)
where

import qualified Data.Foldable as F

import qualified Data.FImage.Image           as Image
import qualified Data.FImage.Image.Simple    as Image.Simple
import qualified Data.FImage.Algebra         as Algebra
import qualified Data.FImage.Transformation  as Transformation
import qualified Data.FImage.Geometry.Vector as Vector

-- translate :: Float -> Float -> Image.Image -> Image.Image
-- translate dx dy = Transformation.translate (Vector.mk dx dy)

scale :: Float -> Float -> Image.Image -> Image.Image
scale dx dy = Transformation.scale (Vector.mk dx dy)

rotate :: Float -> Image.Image -> Image.Image
rotate = Transformation.rotate

diamond :: Float -> Float -> Image.Image
diamond w h = scale (w / sqrt 2) (h / sqrt 2) $ rotate (pi / 4) Image.Simple.uSquare

oval :: Float -> Float -> Image.Image
oval w h = scale (w / sqrt 2) (h / sqrt 2) Image.Simple.uDisk

annulus :: Float -> Float -> Image.Image
annulus f f' = d `Algebra.diff` d'
```

```
  where
    d  = Transformation.uScale f  Image.Simple.uDisk
    d' = Transformation.uScale f' Image.Simple.uDisk

checkerDisk :: Float -> Image.Image
checkerDisk f = Image.Simple.disk f `Algebra.inter` Image.Simple.checker

checkerSquare :: Float -> Image.Image
checkerSquare f = Image.Simple.square f `Algebra.inter` Image.Simple.checker

checkerAnnulus :: Float -> Float -> Image.Image
checkerAnnulus f f' = annulus f f' `Algebra.inter` Image.Simple.checker

octogon :: Float -> Image.Image
octogon f = strip0 `Algebra.inter` strip1 `Algebra.inter` strip2 `Algebra.inter` strip3
  where
    strip0 = Image.Simple.hStrip f
    strip1 = Transformation.rotate (pi / 4) strip0
    strip2 = Transformation.rotate (pi / 4) strip1
    strip3 = Transformation.rotate (pi / 4) strip2

xorCircles2 :: Float -> Float -> Image.Image
xorCircles2 f f' = lDisk `Algebra.xor` rDisk
  where
    disk  = Image.Simple.disk f
    lDisk = Transformation.hTranslate (-f') disk
    rDisk = Transformation.hTranslate f'    disk

checkerXORCircles2 :: Float -> Float -> Image.Image
checkerXORCircles2 f f' = Image.Simple.checker `Algebra.xor` xorCircles2 f f'

xorCircles4 :: Float -> Float -> Image.Image
xorCircles4 f f' = i `Algebra.xor` Transformation.rotate (pi / 2) i
  where
    i  = xorCircles2 f f'

xorCircles8 :: Float -> Float -> Image.Image
xorCircles8 f f' = i `Algebra.xor` Transformation.rotate (pi / 4) i
  where
    i  = xorCircles4 f f'

eclipse :: Float -> Float -> Float -> Image.Image
eclipse f f' f'' = d `Algebra.diff` d'
  where
    d  = Image.Simple.disk f
    d' = Transformation.hTranslate f' $ Transformation.uScale f'' d
```

```haskell
lineCircles :: Float -> Image.Image
lineCircles f = F.foldr1 Algebra.xor ds
  where
    ds = [mk dx | dx <- [-4,-3..4]]
    mk dx = Transformation.uScale (abs dx) $ Transformation.hTranslate dx (Image.Simple.disk f)

timeTunnel :: Image.Image
timeTunnel = F.foldr1 Algebra.xor hps
  where
    hps = [Transformation.rotate (i * pi / 16) $ Image.Simple.vHalfPlane 0 | i <- [1..16]]
```

11 Data.FImage.Image.Simple

```
{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Data.FImage.Image.Simple
(
  -- * Basic boolean images
  hHalfPlane
, hHalfPlane0
, vHalfPlane
, vHalfPlane0
, hStrip
, uHStrip
, vStrip
, uVStrip
, cross
, checker
, altRings
, disk
, uDisk
, square
, uSquare
, polarChecker
, polarDancer
)
where

import qualified Data.FImage.Image                as Image
import qualified Data.FImage.Geometry             as Geometry
import qualified Data.FImage.Geometry.Point       as Point
import qualified Data.FImage.Geometry.PolarPoint as PolarPoint

-- | Horizontal half plane at given y coordinate.
hHalfPlane :: Float -> Image.Image
hHalfPlane y p = Point.y p <= y

-- | Horizontal half plane at y=0 coordinate.
hHalfPlane0 :: Image.Image
hHalfPlane0 = hHalfPlane 0

-- | Vertical half plane at given x coordinate.
vHalfPlane :: Float -> Image.Image
vHalfPlane x p = Point.x p <= x

-- | Horizontal half plane at x=0 coordinate.
vHalfPlane0 :: Image.Image
```

```haskell
vHalfPlane0 = vHalfPlane 0

-- | Infinitely horizontal strip of given width.
hStrip :: Float -> Image.Image
hStrip w p = abs (Point.y p) <= w / 2

-- | Infinitely horizontal strip of unit width.
uHStrip :: Image.Image
uHStrip = hStrip 1

-- | Infinitely vertical strip of given width.
vStrip :: Float -> Image.Image
vStrip w p = abs (Point.x p) <= w / 2

-- | Infinitely horizontal strip of unit width.
uVStrip :: Image.Image
uVStrip = vStrip 1

-- | Infinitely horizontal and vertical strips of given width.
cross :: Float -> Image.Image
cross w p = abs (Point.x p) <= w / 2 || abs (Point.y p) <= w / 2

-- | Checker of unit width.
checker :: Image.Image
checker p = even (x' + y')
  where
    x' = fromIntegral . floor $ Point.x p
    y' = fromIntegral . floor $ Point.y p

-- | Concentric of unit width.
altRings :: Image.Image
altRings p = even . fromIntegral . floor $ Geometry.dist0 p

-- | Disk of given radius.
disk :: Float -> Image.Image
disk r p = Geometry.dist0 p <= r

-- | Disk of unit radius.
uDisk :: Image.Image
uDisk = disk 1

-- | Square of given length.
square :: Float -> Image.Image
square l p = max x y <= l / 2
  where
    x = abs (Point.x p)
```

```
    y = abs (Point.y p)

-- | Square of unit length.
uSquare :: Image.Image
uSquare = square 1.0

-- | Polar checker. The parameter determines the number of alternations, and
-- hence is twice the number of slices.
-- For a cartesian point (x, y), convert to polar point (t', r') and convert
-- back to a cartesian point (r, t * n + pi) and use checker function.
polarChecker :: Float -> Image.Image
polarChecker f = checker . g . Geometry.toPolar
  where
    g p = Point.mk x y
      where
        x = PolarPoint.rho p
        y = f / pi * PolarPoint.theta p

-- | Polar dancer. The parameter determines the "thickness of the dancer's arms".
polarDancer :: Float -> Image.Image
polarDancer f = vStrip f . Geometry.fromPolar . aux . Geometry.toPolar
  where
    aux p = PolarPoint.mk rho' theta'
      where
        rho' = PolarPoint.theta p
        theta' = PolarPoint.rho p
```

12 Data.FImage.Geometry.Point

```haskell
module Data.FImage.Geometry.Point
(
  -- * type
  Point(..)
  -- * constructing
, mk
)
where

-- | type definition
data Point = Point {x :: Float, y :: Float} deriving (Show, Eq, Ord)

-- | Make a point '(x, y)'' from two floats 'x' and 'y'.
mk :: Float -> Float -> Point
mk x y = Point {x = x, y = y}
```

13 Data.FImage.Geometry.PolarPoint

```
module Data.FImage.Geometry.PolarPoint
(
  -- * type
  Angle
, PolarPoint(..)
  -- * constructing
, mk
)
where

-- | PolarPoint type definition
-- Polar coordinates (rho, theta), where rho is the distance
-- from the origin and theta is the angle between
-- the positive x axis and the ray emanating
-- from the origin and passing through the point.
type Angle = Float
data PolarPoint = PolarPoint {rho :: Float, theta :: Angle}
                  deriving (Show, Eq, Ord)

-- | Make a polar point (rho, theta) from distance rho and angle theta.
mk :: Float -> Angle -> PolarPoint
mk rho theta = PolarPoint {rho = rho, theta = theta}

-- | Make a polar point (rho, theta) from distance rho and angle theta.
mk' :: Float -> Angle -> Maybe PolarPoint
mk' rho theta
  | rho < 0   = Nothing
  | otherwise = Just $ PolarPoint {rho = rho, theta = theta}
```

14 Data.FImage.Geometry.Vector

```
module Data.FImage.Geometry.Vector
(
  -- * type definition
  Vector(..)
  -- * constructing
, mk
, mkDiag
  -- * transforming
, revX
, revY
, revXY
, invX
, invY
, invXY
)
where

-- | Vector type definition
data Vector = Vector {x:: Float, y :: Float} deriving (Eq, Ord)

-- | Make a vector (x, y) from two floats `x`and `y`.
mk :: Float -> Float -> Vector
mk x y = Vector {x = x, y = y}

-- | Make a vector '(dz, dz)'' from a float 'dz'.
mkDiag :: Float -> Vector
mkDiag z = mk z z

-- | Make the vector (-x, y) from vector (x, y).
revX :: Vector -> Vector
revX Vector {x  = x, y = y} = mk (-x) y

-- | Make the vector (x, -y) from vector (x, y).
revY :: Vector -> Vector
revY Vector {x  = x, y = y} = mk x (-y)

-- | Make the vector (-x, -y) from vector (x, y).
revXY :: Vector -> Vector
revXY Vector {x  = x, y = y} = mk (-x) (-y)

-- | Make the vector (1/x, y) from vector (x, y).
invX :: Vector -> Vector
invX Vector {x  = x, y = y} = mk (1 / x) y
```

```haskell
-- | Make the vector (x, 1/y) from vector (x, y).
invY :: Vector -> Vector
invY Vector {x  = x, y = y} = mk x (1 / y)

-- | Make the vector (1/x, 1/y) from vector (x, y).
invXY :: Vector -> Vector
invXY Vector {x  = x, y = y} = mk (1 / x) (1 / y)
```

15 Data.FImage.Geometry

```haskell
 module Data.FImage.Geometry
(
  dist
, dist0
  -- * Converting
-- , fromPolar
-- , toPolar
)
where

import qualified Data.FImage.Geometry.Point      as Point
import qualified Data.FImage.Geometry.PolarPoint as PolarPoint

-- | Compute the distance between two cartesian points.
dist :: Point.Point -> Point.Point -> Float
dist Point.Point {Point.x = x1, Point.y = y1}
     Point.Point {Point.x = x2, Point.y = y2} = sqrt $ (x2 - x1)**2 + (y2 - y1)**2

-- | Compute the distance from a given point to the origin.
dist0 :: Point.Point -> Float
dist0 Point.Point {Point.x = x, Point.y = y} = sqrt $ x**2 + y**2

-- | Convert a polar point to a cartesian point.
-- fromPolar :: PolarPoint.PolarPoint -> Point.Point
-- To be implemented...

-- | Convert a cartesian point to a polar point.
-- toPolar :: Point.Point -> PolarPoint.PolarPoint
-- To be implemented...
```

16 Data.FImage.Transformation

```
module Data.FImage.Transformation
(
  -- * type
  Transformation
-- * Transforming boolean images
-- , translate
-- , hTranslate
-- , vTranslate
-- , scale
-- , hScale
-- , vScale
-- , uScale
-- , rotate
)
where

import qualified Data.FImage.Image                as Image
import qualified Data.FImage.Geometry.Point       as Point
import qualified Data.FImage.Geometry.Vector      as Vector

-- | Boolean image transformation type definition.
type Transformation = Image.Image -> Image.Image

-- | Translate a Boolean image according to a vector.
-- translate :: Vector.Vector -> Transformation
-- To be implemented...

-- | Translate a Boolean image horizontally by a given distance.
-- hTranslate :: Float -> Transformation
-- To be implemented...

-- | Translate a Boolean image vertically by a given distance.
-- vTranslate :: Float -> Transformation
-- To be implemented...

-- | Scale a Boolean image according to a vector.
-- scale :: Vector.Vector -> Transformation
-- To be implemented...

-- | Scale a Boolean image horizontally by a given factor.
-- hScale :: Float -> Transformation
-- To be implemented...

-- | Scale a Boolean image vertically by a given factor.
```

```
-- vScale :: Float -> Transformation
-- To be implemented...

-- | Scale a Boolean image uniformly by a given factor.
-- uScale :: Float -> Transformation
-- To be implemented...

-- | Rotate a Boolean image uniformly by a given angle.
-- rotate :: Float -> Transformation
-- To be implemented...
```