Programmation fonctionnelle

Fiche de TP 9

L3 Informatique 2020-2021

Arbres quaternaires

1 Arbres quaternaires

Un *arbre quaternaire* (« *quadtree* » en anglais) est une structure de données très simple pour l'indexation des points dans le plan. Il s'agit d'une structure de données hiérarchique construite par divisions récursives du plan en quatre rectangles disjoints. Plus précisément, dans un arbre quaternaire, chaque sommet représente une partie du plan à indexer, la racine représentant le plan à indexer dans son intégralité. Nous désignerons les parties du plan par des *rectangles*. Chaque sommet est soit une *feuille* contenant zéro, un ou plusieurs points, soit un *sommet interne* ayant exactement quatre fils représentant les quatre rectangles obtenus en divisant la partie du plan associée à ce sommet en deux le long de chaque axe. La figure 1 donne un exemple de division récursive du plan.

Nous avons évoqué le fait qu'une feuille contient zéro, un ou plusieurs points. Un arbre quaternaire est en effet paramétré par un entier positif (noté `n` dans la suite) indiquant le nombre maximal de points pouvant être enregistrés dans une feuille.

Pour faciliter la lecture, nous adopterons les conventions suivantes. Pour un sommet interne, les quatre fils représenteront dans l'ordre

1. le rectangle supérieur gauche ;
2. le rectangle supérieur droit ;
3. le rectangle inférieur droit ;
4. le rectangle inférieur gauche.

Un exemple est donné la figure 2.

Dans ce TP nous allons progressivement développer une implantation des arbres quaternaires en Caml.

Voici à présent une recommandation sur le typage. Le système d'inférence des types du Caml produit le type le plus général possible. Par exemple

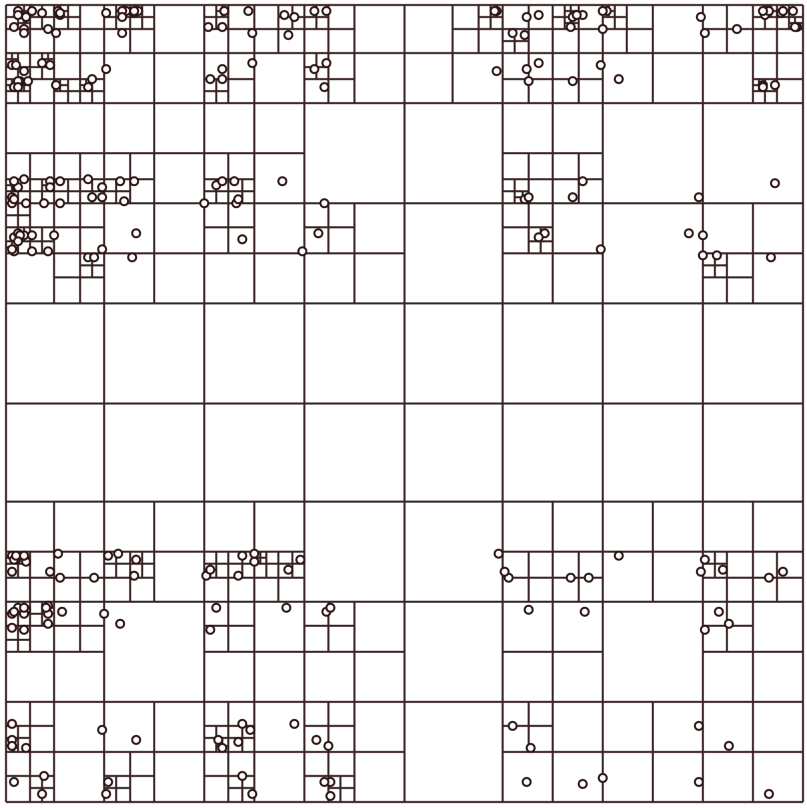

FIGURE 1 – Division récursive du plan offerte par un découpage récursif du rectangle en quatre zones.

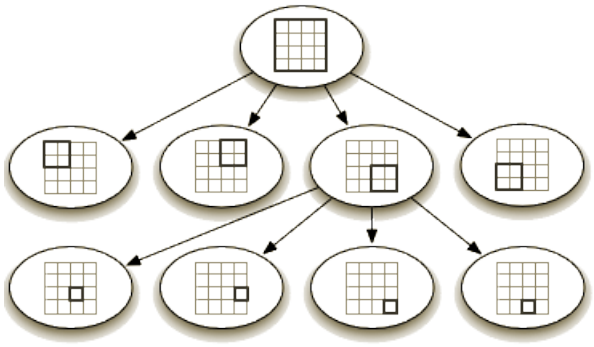

FIGURE 2 – Correspondance entre nœud interne d'un arbre quaternaire et zones d'un rectangle.

```
# type int_pair = int * int;;
type int_pair = int * int

# let p = (1, 2);;
val p : int * int = (1, 2)

# let f = fst;;
val f : 'a * 'b -> 'a = <fun>
```

On peut néanmoins utiliser le typage explicite (syntaxe avec « : ») pour voir apparaître le nom du type désiré. Par exemple,

```
# let p : int_pair = (1, 2);;
val p : int_pair = (1, 2)

# let f : int_pair -> int = fst;;
val f : int_pair -> int = <fun>

# let swap (p : int_pair) : int_pair =
   (snd p, fst p);;
val swap : int_pair -> int_pair = <fun>
```

Il est conseillé de recourir à ces procédés pour faire sorte d'avoir des fonctions qui respectent les types demandés par la spécification du sujet.

2 Points

Les points du plan seront représentés par le type `point` :

```
type point = float * float;;
```

Le type `float` est nécessaire (au détriment du type `int`) puisqu'il sera nécessaire dans la suite de diviser de manière exacte des distances entre des points.

Exercice 1. (Manipulation de points)

1. Écrire la fonction

   ```
   val make_point : float -> float -> point
   ```

   qui renvoie le point dont les coordonnées figurent dans ses deux arguments.

   ```
   # make_point 1.5 2.;;
   - : point = (1.5, 2.)

   # make_point (-1.0) 2.5;;
   - : point = (-1., 2.5)
   ```

2. Écrire les fonctions

   ```
   val point_x : point -> float
   ```

   et

   ```
   val point_y : point -> float
   ```

   qui renvoient respectivement la projection sur l'axe des abscisses ou des ordonnées du point en argument.

```
# let p = make_point 1. 2.;;
val p : point = (1., 2.)

# point_x p;;
- : float = 1.

# point_y p;;
- : float = 2.
```

3. Un point $p_1 := (x_1, y_1)$ *domine* un point $p_2 := (x_2, y_2)$ si $x_1 \geqslant x_2$ et $y_1 \geqslant y_2$. Écrire la fonction

```
val point_domination : point -> point -> bool
```

qui teste si le premier en argument domine le second.

```
# let p1 = make_point 0. 0. and p2 = make_point 0. 1. and p3 = make_point 1. 1.;;
val p1 : point = (0., 0.)
val p2 : point = (0., 1.)
val p3 : point = (1., 1.)

# point_domination p1 p1, point_domination p1 p2, point_domination p1 p3;;
- : bool * bool * bool = (true, false, false)

# point_domination p2 p1, point_domination p2 p2, point_domination p2 p3;;
- : bool * bool * bool = (true, true, false)

# point_domination p3 p1, point_domination p3 p2, point_domination p3 p3;;
- : bool * bool * bool = (true, true, true)
```

3 Rectangles

Un rectangle, dont les cotés sont parallèles aux axes, est défini par deux points du plan : son point inférieur gauche et son point supérieur droit (nous remarquerons que son point supérieur droit domine son point inférieur gauche). Les rectangles seront donc représentés par le type `rectangle` :

```
type rectangle = point * point;;
```

Exercice 2. (Manipulation de rectangles)

1. Écrire la fonction

```
val make_rectangle : point -> point -> rectangle
```

qui renvoie un rectangle construit partir de son point inférieur gauche et de son point supérieur droit. Nous supposerons que le premier point est toujours dominé par le second point lors de l'appel de la fonction (il n'est donc pas nécessaire de faire le test dans la fonction). Il est possible (mais pas obligatoire) de munir cette fonction de la vérification de cette pré-assertion en utilisant `assert`.

```
# let p1 = make_point 0. 0. and p2 = make_point 1. 2.;;
val p1 : point = (0., 0.)
val p2 : point = (1., 2.)

# make_rectangle p1 p2;;
- : rectangle = ((0., 0.), (1., 2.))
```

2. Écrire les fonctions

```
val : rectangle_lower_left : rectangle -> point
```

et

```
val : rectangle_upper_right : rectangle -> point
```

qui renvoient respectivement le point inférieur gauche ou supérieur droit du rectangle en argument.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
val r : rectangle = ((0., 0.), (1., 2.))

# rectangle_lower_left r;;
- : point = (0., 0.)

# rectangle_upper_right r;;
- : point = (1., 2.)
```

3. Écrire les fonctions

```
val : rectangle_width : rectangle -> float
```

et

```
val : rectangle_height : rectangle -> float
```

qui renvoient respectivement la largeur et la hauteur du rectangle en argument.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
val r : rectangle = ((0., 0.), (1., 2.))

# rectangle_width r;;
- : float = 1.

# rectangle_height r;;
- : float = 2.
```

4. Un point $p := (x, y)$ est *contenu* dans un rectangle de point inférieur gauche (x_ℓ, y_ℓ) et de point supérieur droit (x_r, y_r) si $x_\ell \leqslant x \leqslant x_r$ et $y_\ell \leqslant y \leqslant y_r$. Un point situé sur un coté du rectangle ou dans l'un de ses coins est donc contenu dans ce dernier.

   Écrire la fonction

```
val rectangle_contains_point : rectangle -> point -> bool
```

   qui teste si un point en argument est contenu dans le rectangle en argument.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 1.);;
val r : rectangle = ((0., 0.), (1., 1.))

# let p1 = make_point 0. 0.
  and p2 = make_point 0.5 0.5
  and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# rectangle_contains_point r p1;;
- : bool = true

# rectangle_contains_point r p2;;
- : bool = true

# rectangle_contains_point r p3;;
- : bool = false
```

5. Écrire la fonction

```
val rectangle_contained_points : rectangle -> point list -> point list
```

   qui étant donnée un rectangle et une liste de points, renvoie la liste des points qui figurent dans le rectangle. Utiliser obligatoirement une fonction d'ordre supérieur sur les listes.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 1.);;
val r : rectangle = ((0., 0.), (1., 1.))

# let p1 = make_point 0. 0.
  and p2 = make_point 0.5 0.5
  and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# rectangle_contained_points r [p1; p2; p3];;
- : point list = [(0., 0.); (0.5, 0.5)]
```

4 Arbres quaternaires

Pour faciliter l'écriture des fonctions manipulant les arbres quaternaires, nous associerons explicitement à chaque nœud de l'arbre quaternaire le rectangle qui lui est associé. Les arbres quaternaires seront donc représentés par le type *quadtree* :

```
type quadtree =
    |Leaf of point list * rectangle
    |Node of quadtree * quadtree * quadtree * quadtree * rectangle;;
```

Une feuille contient donc l'information sur le rectangle qu'elle représente et les points que celui-ci contient. Un nœud contient un rectangle et quatre sous-arbres, selon les spécifications données au début du document.

Exercice 3. (Manipulation d'arbres quaternaires)

1. Écrire la fonction

   ```
   val rectangle_split_four : rectangle -> rectangle * rectangle * rectangle * rectangle
   ```

   qui renvoie un tuple contenant les les quatre rectangles obtenus en divisant équitablement le long de chaque axe le rectangle en argument.

   ```
   # let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
   val r : rectangle = ((0., 0.), (1., 2.))

   # rectangle_split_four r;;
   - : rectangle * rectangle * rectangle * rectangle =
   (((0., 1.), (0.5, 2.)), ((0.5, 1.), (1., 2.)), ((0.5, 0.), (1., 1.)), ((0., 0.), (0.5, 1.)))
   ```

2. Écrire les fonctions

   ```
   val smallest : 'a list -> 'a
   ```

   et

   ```
   val greatest : 'a list -> 'a
   ```

   qui renvoient respectivement le plus petit et le plus grand élément d'une liste (comparés à l'aide de la fonction polymorphe `compare`). Utiliser obligatoirement une fonction d'ordre supérieur sur les listes.

   ```
   # let l = [2; 4; 6; 1; 8; 4; 3; 1];;
   val l : int list = [2; 4; 6; 1; 8; 4; 3; 1]

   # smallest l;;
   - : int = 1

   # largest l;;
   - : int = 8
   ```

3. Écrire la fonction

```
val enclosing_rectangle : point list -> rectangle
```

qui renvoie le plus petit rectangle contenant tous les points de la liste en argument.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# enclosing_rectangle [p1; p2; p3];;
- : rectangle = ((0., 0.), (0.5, 1.5))
```

4. Écrire la fonction

```
val quadtree_make : point list -> int -> quadtree
```

qui renvoie l'arbre quaternaire correspondant à la liste des points passés en argument. Chaque feuille de l'arbre contient au plus `n` points où `n` est le 2^{e} argument de la fonction.

Un algorithme possible est le suivant :

(a) si le nombre de points restant à insérer est inférieur à `n`, créer une feuille contenant tous les points et la renvoyer ;

(b) sinon, créer un nouveau nœud destiné à représenter un rectangle contenant tous les points de la liste. Disposer ensuite les points de la liste récursivement, en construisant leurs arbres quaternaires associés, dans les quatre sous-arbres du nœud.

(c) Pour la racine, on commence avec le plus petit rectangle contenant tous les points.

```
# let p1 = make_point 0. 0.
  and p2 = make_point 0.5 0.5
  and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# quadtree_make [p1; p2; p3] 1;;
- : quadtree =
Node (Leaf ([], ((0., 0.75), (0.25, 1.5))),
 Leaf ([(0.5, 1.5)], ((0.25, 0.75), (0.5, 1.5))),
 Leaf ([(0.5, 0.5)], ((0.25, 0.), (0.5, 0.75))),
 Leaf ([(0., 0.)], ((0., 0.), (0.25, 0.75))), ((0., 0.), (0.5, 1.5)))
```

5. Écrire la fonction

```
val quadtree_count : quadtree -> int
```

qui renvoie le nombre de points enregistrés dans l'arbre quaternaire en argument.

```
# let p1 = make_point 0. 0.
  and p2 = make_point 0.5 0.5
  and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# quadtree_count (quadtree_make [p1; p2; p3] 1);;
- : int = 3
```

6. Écrire la fonction

```
val quadtree_signature : quadtree -> int list
```

qui renvoie la liste des nombres de points contenus dans chaque feuille de l'arbre quaternaire en argument. L'arbre est parcouru de gauche à droite et la liste renvoyée et construite selon cet ordre.

```
# let p1 = make_point 0. 0.
  and p2 = make_point 0.5 0.5
  and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# quadtree_signature (quadtree_make [p1; p2; p3] 1);;
- : int list = [0; 1; 1; 1]
```

5 Utilisation des arbres quaternaires

Nous allons maintenant utiliser tout ce qui a été mis en place précédemment pour savoir si un arbre quaternaire possède un point particulier.

Exercice 4. (Recherche de points)

1. Écrire la fonction

```
val quadtree_all_points : quadtree -> point list
```

qui renvoie la liste de tous les points contenus dans les feuilles de l'arbre quaternaire en argument.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)

# quadtree_all_points (quadtree_make [p1; p2; p3] 1);;
- : point list = [(0.5, 1.5); (0.5, 0.5); (0., 0.)]
```

On suppose données les fonctions

(a)
```
val rectangle_contains_rectangle : rectangle -> rectangle -> bool
```

qui teste si un rectangle est totalement inclus dans un autre ;

(b)
```
val rectangle_disjoint_rectangle : rectangle -> rectangle -> bool
```

qui teste si deux rectangles donnés ont au moins un point d'intersection ;

(c)
```
val rectangle_intersection_rectangle :
    rectangle -> rectangle -> rectangle
```

qui renvoie le rectangle à l'intersection de deux rectangles.

Des implantations possibles sont

```
let rectangle_contains_rectangle r1 r2 =
    rectangle_contains_point r1 (rectangle_lower_left r2)
    && rectangle_contains_point r1 (rectangle_upper_right r2);;

let rectangle_disjoint_rectangle r1 r2 =
    let r1_lower_left = rectangle_lower_left r1 in
    let r1_upper_right = rectangle_upper_right r1 in
    let r2_lower_left = rectangle_lower_left r2 in
    let r2_upper_right = rectangle_upper_right r2 in
    point_domination r2_lower_left r1_upper_right
    || point_domination r1_lower_left r2_upper_right;;

let rectangle_intersection r1 r2 =
    let r1_ll = rectangle_lower_left r1 in
    let r1_ur = rectangle_upper_right r1 in
    let r2_ll = rectangle_lower_left r2 in
    let r2_ur = rectangle_upper_right r2 in
    let ll_x_max = largest [point_x r1_ll; point_x r2_ll] in
    let ll_y_max = largest [point_y r1_ll; point_y r2_ll] in
    let ur_x_min = smallest [point_x r1_ur; point_x r2_ur] in
    let ur_y_min = smallest [point_y r1_ur; point_y r2_ur] in
    let ll = make_point ll_x_max ll_y_max in
    let ur = make_point ur_x_min ur_y_min in
    make_rectangle ll ur;;
```

En utilisant ces fonctions, écrire la fonction

```
val quadtree_rectangle_query :rectangle -> quadtree -> point list
```

qui renvoie la liste des points enregistrés dans l'arbre quaternaire en argument qui sont contenus dans le rectangle en argument. Le calcul doit être **efficace**. Par efficace, nous entendons

— qu'il n'est pas nécessaire de parcourir un arbre quaternaire si le rectangle associé à sa racine n'a aucun point commun avec le rectangle requête,

— que tous les points contenus dans les feuilles d'un arbre quaternaire sont nécessairement des solutions si le rectangle associé à la racine de cet arbre quaternaire est inclus dans le rectangle requête.

Nous terminons ce TP en réalisant un support pour l'ajout et la suppression de points dans les arbres quaternaires.

Exercice 5. (Ajout et suppression de points)

1. Écrire la fonction

```
val quadtree_insert : quadtree -> int -> point -> quadtree
```

qui, étant donné un arbre quaternaire contenant au plus `n` points dans chaque feuille renvoie le nouvel arbre quaternaire obtenu en insérant le point en argument. L'arbre renvoyé doit contenir lui aussi au plus `n` points dans chaque feuille.

2. Écrire la fonction

```
val quadtree_delete : quadtree -> point -> quadtree
```

qui, étant donné un arbre quaternaire renvoie le nouvel arbre quaternaire obtenu en supprimant le point donné en argument. Ce point peut exister ou non dans l'arbre d'origine.

Annexe

```
# let points = [ (make_point 0. 0.); (make_point 0.1 0.2); (make_point 0.01 0.01); (make_point 2.
    0.5); (make_point 0.5 1.5)];;
val points : point list =
  [(0., 0.); (0.1, 0.2); (0.01, 0.01); (2., 0.5); (0.5, 1.5)]

# let r = enclosing_rectangle points;;
val r : rectangle = ((0., 0.), (2., 1.5))

# let qt = quadtree_make points 2;;
val qt : quadtree =
  Node (Leaf ([(0.5, 1.5)], ((0., 0.75), (1., 1.5))),
   Leaf ([], ((1., 0.75), (2., 1.5))),
   Leaf ([(2., 0.5)], ((1., 0.), (2., 0.75))),
   Node (Leaf ([], ((0., 0.375), (0.5, 0.75))),
    Leaf ([], ((0.5, 0.375), (1., 0.75))),
    Leaf ([], ((0.5, 0.), (1., 0.375))),
    Node (Leaf ([(0.1, 0.2)], ((0., 0.1875), (0.25, 0.375))),
     Leaf ([], ((0.25, 0.1875), (0.5, 0.375))),
     Leaf ([], ((0.25, 0.), (0.5, 0.1875))),
     Leaf ([(0., 0.); (0.01, 0.01)], ((0., 0.), (0.25, 0.1875))),
```

```
    ((0., 0.), (0.5, 0.375))),
   ((0., 0.), (1., 0.75))),
  ((0., 0.), (2., 1.5)))

# let count = quadtree_count qt;;
val count : int = 5

# let signature = quadtree_signature qt;;
val signature : int list = [1; 0; 1; 0; 0; 0; 1; 0; 0; 2]

# let all_points = quadtree_all_points qt;;
val all_points : point list =
 [(0.5, 1.5); (2., 0.5); (0.1, 0.2); (0., 0.); (0.01, 0.01)]


# let rec range i j = if i > j then [] else i::(range (i+1) j);;
val range : int -> int -> int list = <fun>

# let cartesian l l' =
 List.concat (List.map (fun e -> List.map (fun e' -> (e, e')) l') l);;
val cartesian : 'a list -> 'b list -> ('a * 'b) list = <fun>

# let points m n =
 let xs = List.map float_of_int (range 0 m) and
     ys = List.map float_of_int (range 0 n) in
  List.map (fun (x, y) -> make_point x y ) (cartesian xs ys);;
val points : int -> int -> point list = <fun>

# let ps = points 4 4
val ps : point list =
 [(0., 0.); (0., 1.); (0., 2.); (0., 3.); (0., 4.); (1., 0.); (1., 1.);
  (1., 2.); (1., 3.); (1., 4.); (2., 0.); (2., 1.); (2., 2.); (2., 3.);
  (2., 4.); (3., 0.); (3., 1.); (3., 2.); (3., 3.); (3., 4.); (4., 0.);
  (4., 1.); (4., 2.); (4., 3.); (4., 4.)]

# let qt = quadtree_make ps 2;;
val qt : quadtree =
 Node
  (Node
    (Node (Leaf ([(0., 4.)], ((0., 3.5), (0.5, 4.))),
     Leaf ([(1., 4.)], ((0.5, 3.5), (1., 4.))),
     Leaf ([(1., 3.)], ((0.5, 3.), (1., 3.5))),
     Leaf ([(0., 3.)], ((0., 3.), (0.5, 3.5))), ((0., 3.), (1., 4.))),
    Node (Leaf ([(1., 4.)], ((1., 3.5), (1.5, 4.))),
     Leaf ([(2., 4.)], ((1.5, 3.5), (2., 4.))),
     Leaf ([(2., 3.)], ((1.5, 3.), (2., 3.5))),
     Leaf ([(1., 3.)], ((1., 3.), (1.5, 3.5))), ((1., 3.), (2., 4.))),
    Node (Leaf ([(1., 3.)], ((1., 2.5), (1.5, 3.))),
     Leaf ([(2., 3.)], ((1.5, 2.5), (2., 3.))),
     Leaf ([(2., 2.)], ((1.5, 2.), (2., 2.5))),
     Leaf ([(1., 2.)], ((1., 2.), (1.5, 2.5))), ((1., 2.), (2., 3.))),
```

```
    Node (Leaf ([(0., 3.)], ((0., 2.5), (0.5, 3.))),
     Leaf ([(1., 3.)], ((0.5, 2.5), (1., 3.))),
     Leaf ([(1., 2.)], ((0.5, 2.), (1., 2.5))),
     Leaf ([(0., 2.)], ((0., 2.), (0.5, 2.5))), ((0., 2.), (...))),
    ...),
  ...)

# let points_in = let p1 = make_point 0. 0. and p2 = make_point 2. 3. in
   let query = make_rectangle p1 p2 in
   quadtree_rectangle_query query qt;;
val points_in : point list =
  [(1., 3.); (0., 3.); (2., 3.); (2., 2.); (1., 2.); (0., 2.); (2., 1.);
   (2., 0.); (1., 1.); (0., 1.); (1., 0.); (0., 0.)]

# let points_in = let p1 = make_point 3. 3. and p1 = make_point 5. 5. in
   let query = make_rectangle p1 p2 in
   quadtree_rectangle_query query qt;;
val points_in : point list = [(3., 4.); (3., 3.); (4., 4.); (4., 3.)]

# let points_in = let p1 = make_point 2. (-1.) and p2 = make_point 3. 5. in
   let query = make_rectangle p1 p2 in
   quadtree_rectangle_query query qt;;
val points_in : point list =
  [(2., 4.); (2., 3.); (2., 2.); (3., 4.); (3., 3.); (3., 2.); (3., 1.);
   (2., 1.); (3., 0.); (2., 0.)]

# let points_in = let p1 = make_point 3.75 3.75 and p2 = make_point 6. 6. in
   let query = make_rectangle p1 p2 in
   quadtree_rectangle_query query qt;;
val points_in : point list = [(4., 4.)]

# let points_in = let p1 = make_point 4.75 4.75 and p2 = make_point 6. 6. in
   let query = make_rectangle p1 p2 in
   quadtree_rectangle_query query qt;;
val points_in : point list = []
```