Programmation avancée

TD2 : Modules

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse

Dans la partie 3, nous aurons besoin de la bibliothèque externe `ocamlgraph`. Vous pouvez dès maintenant l'installer avec `opam` :

```
opam install ocamlgraph
```

Téléchargez ensuite le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/02-modules/td02.zip`. Extraire l'archive `.zip`. Utiliser `dune` pour compiler le projet et exécuter les programmes. Le code à compléter est dans le dossier `bin/`, avec un fichier par partie.

1 Préliminaires

On commence ce TP par quelques exercices pour s'exercer et s'habituer à la syntaxe du langage de modules et signatures d'OCaml.

On considère la signature de module `Counter` ci-dessous, qui correspond à la signature d'un module de « compteurs » : chaque valeur de type `t` correspond à un compteur mutable, initialisé à zéro, que l'on peut incrémenter, et dont on peut demander la valeur courante.

```
module type Counter = sig
  type t
  val create : unit -> t
  val increment : t -> unit
  val current_value : t -> int
end
```

Exercice 1. *Écrire un module* `MyCounter` *implémentant la signature* `Counter` *(*`module MyCounter : Counter = ...`*).*
Dans le code suivant :

```
let c = MyCounter.create () in
MyCounter.increment c;
MyCounter.increment c;
(* code sans appel à MyCounter.increment *)
...
let x = MyCounter.current_value c in
x
```

Peut on affirmer que la valeur de `x` *est 2 ? Pourquoi ?*
Pourrait-on en dire autant si on enlevait la contrainte : `module MyCounter = ...` *?*

Exercice 2. *Définir une signature de module* `Ordered`*, déclarant un type* `t` *et une fonction de comparaison* `compare` *de type* `t -> t -> int`*. (Par convention,* `compare` *renvoie* -1 *pour le cas « plus petit que », 0 en cas d'égalité, et 1 dans le cas « plus grand que ».)*

Exercice 3. *Faire de même avec une signature* `Eq` *déclarant un type* `t` *et une fonction* `equal` *de type* `t -> t -> bool`*.*

Exercice 4. *Considérons le module* `String` *de la bibliothèque standard, dont voici le lien vers la documentation :* `https://v2.ocaml.org/api/String.html`*. Satisfait-il la signature* `Ordered` *? Et la signature* `Eq` *? Qu'en est-il du module* `List` *(*`https://v2.ocaml.org/api/List.html`*) ? Justifier.*

Exercice 5. *On écrit* `module M = (String : Ordered)`*.*

1. *Quelle est la signature du module* `M` *? La fonction* `String.length` *existe, mais peut-on appeler* `M.length` *?*
2. *Peut-on utiliser* `M.compare` *pour comparer deux chaines de caractères, et pourquoi ? (par exemple* `M.compare "abc" "def"`*)*
3. *Comment définir* `M` *à partir de* `String` *pour préserver l'égalité de type entre* `M.t` *et* `string`*, tout en n'exposant que la fonction* `compare` *? (indice : utiliser le mot-clef* `with`*…)*

La signature `Ordered` correspond à la signature `OrderedType` définie dans les modules `Set` et `Map` de la bibliothèque standard. Ces modules fournissent un foncteur `Make` pour construire respectivement une structure d'ensemble et de table associative à partir d'un module implémentant `OrderedType` (documentation : `https://v2.ocaml.org/api/Set.html` et `https://v2.ocaml.org/api/Map.html`).

Exercice 6. *Définir un module* `Couleur` *contenant un type* `t` *:*

```
type t = { vert : int; bleu : int; rouge : int }
```

et une fonction de comparaison associée (de type `t -> t -> int`*).*

1. *Utiliser le foncteur* `Set.Make` *pour obtenir un module* `CouleurSet` *implémentant des ensembles de couleurs.*
2. *Utiliser maintenant* `Map.Make` *pour construire un module* `CouleurSetMap` *fournissant des tables associatives dont les clefs sont des* `CouleurSet.t` *(donc des ensembles de couleurs).*
3. *En utilisant le module* `CouleurSetMap` *et les liens vers la documentation de* `Set` *et* `Map` *ci-dessus, écrire une fonction* `make_couleur_map` *qui crée une table associative, associant l'ensemble { couleur verte; couleur bleue } à l'entier 42. (Avec couleur verte =* `{vert = 255; bleu = 0; rouge = 0}` *et de même pour les autres couleurs.)*

Exercice 7. *Implémenter un foncteur* `Lexico`*, prenant en entrée un module satisfaisant la signature* `Ordered`*, et produisant en sortie un module satisfaisant la signature* `Ordered` *pour des listes d'éléments du module d'entrée, comparées par ordre lexicographique.*

Voici un exemple d'utilisation du module `Lexico`, qui doit fonctionner avec votre implémentation :

```
module M1 = Lexico(Int)
module M2 = Lexico(String)
module M3 = Lexico(Lexico(Int))

let () =
  assert (M1.compare [0; 2] [1; 2] = -1);
  assert (M1.compare [1; 2; 4] [1; 3] = -1);
  assert (M1.compare [0; 1; 2] [0; 1] = 1);
  assert (M1.compare [] [] = 0);
  assert (M1.compare [] [0;1] = -1)
  assert (M2.compare ["a"] ["b"; "c"] = -1);
  assert (M2.compare ["b"; "d"] ["b"; "c"] = 1);
  assert (M3.compare [] [[]] = -1);
  assert (M3.compare [[0]] [[]] = 1)
```

2 Monoïdes

En mathématiques, un monoïde est une structure algébrique définie comme un ensemble muni d'une loi de composition associative et d'un élément neutre. En OCaml, ceci correspond à un type (le type des éléments de l'ensemble du monoïde), d'une fonction à deux arguments permettant de combiner deux éléments de ce type et en produire un autre, et une valeur de ce type pour l'élément neutre.

Exercice 8. *Définir une signature de module OCaml* `Monoide` *correspondant à un monoïde. On nommera* `t` *le type des éléments du monoïde,* `op` *la fonction de composition du monoïde, et* `e` *l'élément neutre. Pour des raisons de débogage, il est pratique d'ajouter une fonction* `to_string` *à la signature, qui définit comment convertir un élément du monoïde en chaîne de caractère pour l'affichage.*

Exercice 9. *Peut-on utiliser le système de types d'OCaml pour garantir que la fonction de composition* `op` *soit effectivement associative et que* `e` *soit effectivement un élément neutre vis-à-vis de* `op` *?*

Exercice 10. *Définir un module OCaml* `NatMon` *implémentant la structure de monoïde correspondant au monoïde* $(\mathbb{N}, +, 0)$ *(il doit respecter votre signature de la question précédente). (Ici,* $\mathbb{N}$ *est l'ensemble des éléments du monoïde,* $+$ *la loi de composition, et* 0 *l'élément neutre. On ignorera les questions de débordement arithmétiques et on considérera que* $\mathbb{N}$ *est implémentable par le type OCaml* `int`*.)*

Exercice 11. *Définir un type OCaml dont les valeurs correspondent aux éléments de l'ensemble* $\overline{\mathbb{N}} \triangleq \mathbb{N} \cup \infty$.

Exercice 12. *Définir un module OCaml* `NatBarMon` *implémentant la structure de monoïde* $\left(\overline{\mathbb{N}}, \min, \infty\right)$.

Exercice 13. *À partir d'un monoïde* $(M, \oplus, e)$*, on peut considérer le monoïde des matrices carrées de taille* n *(pour* n *donné) à valeurs dans* M*, avec comme loi de composition l'addition matricielle, et pour élément neutre la matrice nulle.*

Implémenter un foncteur OCaml `MatMon`*, prenant en arguments :*

- *un module satisfaisant la signature de monoïde*
- *un module spécifiant un entier* n

produit en retour un module implémentant le monoïde des matrices carrées de taille n *à valeurs dans le monoïde d'entrée.*

3 Plus court chemin avec ocamlgraph

Dans cette partie, notre objectif sera d'utiliser la bibliothèque de graphes `ocamlgraph` et son implémentation d'un algorithme de calcul de plus court chemin, afin de trouver notre chemin dans des labyrinthes et autres terrains en 2D.

Par exemple, étant donné le labyrinthe ci-dessous, où `X` représente un mur, on cherche à aller de la position de départ `S` à la position d'arrivée `E`, et ce de manière optimale.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X S     X         X X         X
X   XX  X  XXXXX    X   XXXX  X
X   XXX   X    X  X     X  X  X
X   X    X  XXXX  X  X     X  X
X   X   X   X XX     XXXX XXX X
X   XXXX    X   XX XX  XX     X
X             X        X    E X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

On se munira de la documentation en ligne d'`ocamlgraph` (`http://ocamlgraph.lri.fr/doc` et/ou `https://ocaml.org/p/ocamlgraph/2.0.0/doc/Graph/index.html`) pour voir les modules et fonctions disponibles. (**NB: les modules listés sur cette page habitent tous dans le module `Graph`.** Ainsi, on écrira par exemple `Graph.Imperative` pour accéder au module `Imperative` documenté sur cette page.)

Exercice 14. *En utilisant `ocamlgraph` et en instanciant le bon foncteur, créer le module des graphes non-orientés dont les sommets sont des paires d'entiers (`int * int`, correspondant à des coordonnées sur le terrain) et dont les arêtes sont étiquetés par des `int`. (Regarder du côté du module `Imperative` d'`ocamlgraph`.) On appellera ce module `TGraph`.*

Le code de démarrage fourni avec le TP pour cette partie (`bin/partie3.ml`) contient la définition de quelques terrains de départ que l'on pourra utiliser pour faire des tests (libre à vous de définir des terrains supplémentaires).

Par ailleurs, le code fourni avec le TP contient une petite bibliothèque permettant de lire et travailler sur ces terrains : il s'agit du module `Terrain`. Ouvrir le fichier interface correspondant : `lib/terrain.mli`, et lire la documentation du module.

Exercice 15. *Étant donné une valeur de type `Terrain.t`, en utilisant les fonctions d'`ocamlgraph` et du module `Terrain`, implémenter une fonction `graph_of_terrain` de type `Terrain.t -> TGraph.t`.*

L'objectif est d'obtenir un graphe dont les nœuds correspondent aux positions sur le terrain, connectés par une arête si on peut se déplacer d'une position à l'autre de proche en proche. Le graphe renvoyé par cette fonction doit donc contenir une arête entre les positions `(x, y)` et `(x', y')` si et seulement si :

- *ces deux positions sont voisines sur le terrain (les diagonales ne comptent pas) ;*
- *on peut effectivement se déplacer de l'une position à l'autre : l'une d'entre elles n'est pas un mur, ou en dehors du terrain ;*
- *la valeur de chaque arête est fixée à 1 pour le moment.*

`ocamlgraph` contient une implémentation de l'algorithme de Dijkstra permettant de calculer le plus court chemin entre deux nœuds dans un graphe (Regarder du côté du module `Path`).

Exercice 16. *Instancier le foncteur implémentant l'algorithme de Dijkstra sur votre module de graphe. Implémenter une fonction* `chemin` *de type* `Terrain.t -> (int * int) list` *calculant le plus court chemin entre la position de départ et d'arrivée pour le terrain donné en entrée.*
Utiliser `Terrain.print_with_path` *pour afficher et vérifier le résultat de la fonction sur les terrains d'exemple.*

Les cases d'un terrain peuvent en fait être d'un autre type que « mur » ou « vide » : certaines cases sont des marais, représentées graphiquement par des cases `%`. Il est possible de traverser les cases « marais », mais cela prend deux fois plus de temps que traverser une case vide.

Exercice 17. *Modifier la valeur des arêtes du graphe renvoyé par* `graph_of_terrain`*. Désormais, se déplacer depuis une case marais ou vers une case marais correspond à une arête de valeur 2, contrairement aux arêtes de valeur 1 pour les déplacements sur les cases vides.*

Exercice 18. *Fabriquer un terrain comportant un marais, et illustrer que le malus de déplacement des marais est bien pris en compte par la fonction* `chemin` *: étant donné deux chemins possibles, l'un étant légèrement plus court mais comportant des marais, un chemin un peu plus long sans marais devrait être choisi.*
Écrire du code OCaml affichant le chemin calculé pour votre terrain ainsi fabriqué (en appelant les fonctions de `Terrain` *et votre code).*

Rustlings pour la semaine prochaine

Aller jusqu'à `06_move_semantics` inclus.