Programmation avancée

TD10 : Monades

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `failwith "TODO"`.

Pensez également à tester votre implémentation avec `dune test`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/10-monades/td10.zip`

Pour ce qui suit, vous pourrez à votre convenance, définir du sucre syntaxique pour les opérateurs de binding : `let (let*) = (>>=)`, et ainsi écrire `let* x = monadic_value in return @@ f x` plutôt que `monadic_value >>= (fun x -> return @@ f x)`

1 Monade d'exception

Fichier `src/exception.ml`

La monade d'exception est basée sur

```
type 'a t = Val of 'a | Exn of exn
```

L'idée est qu'un calcul avec exception peut soit retourner normalement une valeur `v`, auquel cas il sera représenté par `Val v`, soit retourner en levant une exception `e`, auquel cas il sera représenter par `Exn e`.

Exercice 1. *Implémenter la monade d'exception comme un module `Exn` ayant la signature `EXN`, définie ci-dessous :*

```
module type MONAD = sig
  type 'a t
  val return : 'a -> 'a t
  val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
end

module type EXN = sig
  include MONAD
  val throw : exn -> 'a t
```

```
  val try_with : 'a t -> (exn -> 'a t) -> 'a t
  val run : 'a t -> 'a option
  (* renvoie None dans le cas d'une exception non rattrapée *)
end
```

Bien penser à décommenter les tests inclus dans le code de démarrage.

2 Monade de *parsing*

Fichier `src/parsing.ml`

La monade de parsing combine la monade de lecture (on lit un flux de tokens) et la monade de non-déterminisme (par défaut, une grammaire est ambiguë et le parsing peut avoir plusieurs résultats). Le type de la monade est

`'a t = char list -> (char list * 'a) list`.

Les calculs représentés (des parseurs) prennent en entrée une suite de tokens de type char et renvoient une liste d'alternatives, chacune étant composée:

- d'un suffixe de la liste d'entrée représentant les tokens non lus ;
- du résultat du parsing.

Pour fixer les idées, cette monade va nous permettre d'écrire des parseurs et de les composer. Par exemple le parseur `int` lira un maximum de caractères de `[0-9]` dans le flux de tokens, et renverra (avec la liste de tokens restante) leur interprétation comme un `int`. Le parseur `plus` pourra ensuite exécuter le parseur d'entiers, récupérer son résultat `n`, puis lire le caractère `+` et relancer le parseur d'entiers, récupérer son résultat `m`, et renvoyer `n + m`. On notera ici deux points :

- les caractères consommés dans la `char list` par un calcul de la monade ne sont plus disponibles pour les calculs suivants ;
- dans cet exemple on retourne au plus un résultat, mais la structure de liste permet de modéliser les échecs (par exemple, sur l'entrée `123++`).

Exercice 2. *Construire un module `Parsing` de signature `MONAD` de telle sorte que les opérations `return` et `bind` respectent la sémantique attendue :*

- *`return` ne consomme pas de token, et ne renvoie qu'un résultat possible;*
- *`bind m f` va renvoyer chaque résultat possible de `f x` où `x` est un résultat possible de `m`, en chaînant les listes de tokens de sorte que les tokens soient consommés par `m` puis `f`.*

Exercice 3. *Étendre votre module, et lui attacher la signature suivante:*

```
module type PARSING = sig
  include MONAD
  (** choix non-déterministe entre deux calculs *)
  val plus : 'a t -> 'a t -> 'a t
  (** l'échec (aucun résultat possible) *)
```

```
  val zero : 'a t
  (** lit (et consomme) le premier char du flux d'entrée, échoue si le flux
est déjà terminé *)
  val read : char t
  (** end of stream, indique si le flux est terminé *)
  val eos  : bool t
  (** exécute un calcul sur une entrée donnée *)
  val run  : 'a t -> char list -> (char list * 'a) list
end
```

2.1 Regexp

Fichier `src/regexp.ml`

L'objectif, à partir de maintenant, est de ne plus jamais s'appuyer sur la définition de `'a Parsing.t`, mais d'utiliser uniquement les opérations élémentaires définies ci-dessus. (Techniquement, on s'appuiera sur le fait que le type `'a t` est abstrait dans la signature `PARSING` et qu'on travaille désormais en dehors de ce module.)

On pose `type word = char list`.

Nous allons définir des combinateurs de parsing qui permettent notamment de construire, pour tout langage régulier L, un parseur $p(L)$ de type `word t` tel que, pour tout préfixe w du flux d'entrée, $p(L)$ lit w et renvoie (au moins) une fois w ssi w appartient à L. L'opération `zero : word t` correspond déjà au langage vide selon cette spécification, et `plus` correspond à l'union de langages.

Exercice 4. *Définir `let epsilon : word t` correspondant au langage contenant uniquement le mot vide: epsilon ne doit donc rien lire et juste renvoyer [].*

Exercice 5. *Définir `let char (c : char) : word t` tel que `char c` reconnaisse le mot restreint au caractère `c` le calcul doit donc vérifier qu'il y a un caractère à lire, et qu'il s'agit bien de `c`.*

Exercice 6. *Définir `let concat (l1 : word t) (l2 : word t) : word t` qui corresponde à la concaténation de langages.*

Exercice 7. *Utiliser les opérations précédemment définies pour dériver le parseur reconnaissant une chaîne donnée: `let string (s : string) : word t`.*

Exercice 8. *Définir `let star (l : word t) : word t` en suivant l'équation $A* = \varepsilon \mid (A + A*)$. Il faudra faire attention aux récursions, et introduire si besoin une notion de concaténation paresseuse qui ne calcule son second parseur qu'une fois qu'on a obtenu un résultat du premier parseur.*

2.2 Expressions

Fichier `src/expr.ml`

Un parseur ne doit pas seulement servir à reconnaître des (sous-)mots, mais aussi à les structurer ou les interpréter. Nous allons donc arrêter de considérer uniquement des objets de type `word t`.

Exercice 9. *Définir* `let digit : int t` *qui reconnaît les mots composés d'un seul chiffre et renvoie leur valeur numérique. En déduire un parseur* `let expr : int t` *qui reconnaît les expressions arithmétiques simples définies ci-dessous et renvoie leur interprétation dans* `int` *:*

```
digit ::= ['0'-'9']
expr  ::= digit | digit '+' expr
```

Exercice 10. *Que se passe-t-il si on écrit le même langage par une grammaire récursive à gauche?*

On pourra enfin ajouter la multiplication en respectant les priorités usuelles, du parenthésage, lire des entiers supérieurs à 9, etc.

3 Monade de non-déterminisme

Le code de démarrage fourni contient une implémentation simple d'une monade de non-déterminisme, dont la signature `NONDET` est reproduite ci-dessous.

```
module type NONDET = sig
  include MONAD

  val choice : 'a t list -> 'a t
  (** Choix non-déterminisme entre zéro, une ou plusieurs
      possibilités. *)
  val fail : 'a t
  (** Échec. Équivalent à [choice []] : le choix entre zéro
      possibilités. *)
  val either : 'a t -> 'a t -> 'a t
  (** Choix entre deux possibilités. [either a b] est équivalent à
      [choice [a;b]]. *)
  val (|||) : 'a t -> 'a t -> 'a t
  (** Opérateur infixe correspondant à [either]. *)
  val run : 'a t -> 'a list
  (** [run m] exécute le calcul monadique [m] et renvoie la liste des
      valeurs possibles. *)
end
```

Un calcul monadique de type `'a Nondet.t` correspond à calculer, de façon non-déterministe, un certain nombre de valeurs de type `'a` (potentiellement zéro, auquel cas le calcul échoue). L'implémentation fournie implémente naïvement cette interface en

prenant `'a Nondet.t = 'a list`, ce qui correspond à effectivement calculer toutes les valeurs potentielles possibles.

Le but de cet exercice est d'utiliser cette monade pour résoudre le puzzle suivant :

C'est la nuit noire et 4 personnes (Alice, Bob, Carla et Dom) se trouvent bloquées sur une des berges de la rivière où se trouve un pont suspendu. Nos aventuriers sont équipés d'une seule torche. Par ailleurs, on sait qu'Alice met 1 minute pour traverser le pont, Bob en met 2, Carla 5 et Dom 10. Pour traverser le pont, ils sont obligés de s'équiper de la torche et le pont supporte au maximum deux personnes. De plus, si deux personnes traversent le pont en même temps, elles iront au rythme de la personne la plus lente. Peuvent-ils traverser le pont en moins de 17 minutes ?

Exercice 11. *Fichier `src/puzzle.ml`*

Partir du code de démarrage fourni et résoudre le puzzle (remplir la fonction `let solution : trace t`). Il y a exactement deux réponses possibles.

On pourra tester l'affichage de la solution avec `dune exec src/puzzle.exe`.

4 Bonus: Monade de sélection

Fichier `src/select.ml`

On se propose de résoudre le problème n-queens en utilisant une monade un poil exotique: la monade de sélection, de type `'a SelectMonad.t = ('a -> bool) -> 'a option`. Cette monade modélise la recherche d'un élément vérifiant un prédicat. On appelle *sélection* ces valeurs monadiques.

Concrètement, passer un prédicat `p: 'a -> bool` à une sélection de type `'a SelectMonad.t` renvoie une valeur qui satisfait le prédicat; et `None` si il n'en existe pas. Pour notre exemple de n-queens, on pourra se donc se contenter d'utiliser un prédicat vérifiant si un échiquier est correct, et le passer à une selection de type `int list SelectMonad.t`.

Exercice 12. *Définir les opérations `return` et `>>=`.*

On représente un échiquier $n * n$ par une liste de taille n, donc la position de chaque élément et sa valeur indiquent respectivement les positions horizontales et verticales d'une reine sur l'échiquier. Les fonctions de validation d'échiquier sont fournies mais vous ne devriez pas avoir à les manipuler.

Exercice 13. *Définir une fonction `element_select : 'a list -> 'a SelectMonad.t` qui renvoie une sélection renvoyant `Some x` si `x` si satisfait le prédicat et `None` sinon.*

Exercice 14. *Définir une fonction `permutation_select: 'a list -> 'a list SelectMonad.t` qui sélectionne une permutation dans une liste.*

Pour votre information, la monade de sélection peut manipuler autre chose que des prédicats : on pourrait utiliser des sélections de type `type 'a SelectMonad.t = ('a -> r) -> 'a` où `r` est un type arbitraire, par exemple des valeurs dans un espace probabilisé.