Principes de Programmation
Cinquième Phase du Projet de Programmation

9 mai 2019

Dans la dernière phase, vous avez définit les automates finis et les automates finis non déterministes. Mais dans la littérature, il existe des dizaines d'autres variations des automates finis. Nous en décrivons quelques unes (pour fournir un contexte), puis nous expliquons pourquoi ces variations peuvent être vues comme des automates finits dans des environnements monadiques différents. Après quoi nous expliqueront comment rendre nos automate finit "monadique".

Ce sujet est très long, mais uniquement car il donne du contexte, lisez le en diagonal la première fois, car il y a peu d'éléments qui sont nécéssaire pour les questions concrètes de la phase, qui sont décrites Section 4.

Attention, dans ce projet, nous utilisons des monades particulières appelées `MonadFail`. Une monadFail est une monade avec une valeur d'échec `fail::String -> m a`, comme Maybe par exemple. Pour des raisons historiques, dans Haskell, il n'y a pas de différence entre une Monad et une MonadFail et la type-class de base de monad est en fait une monadFail, mais c'est un défaut de Haskell qui devrait être réparer dans une version future. Voire Section 6 pour plus d'informations.

1 Divers variantes d'automates finis

Automates à compteur. Les automates à compteur sont des automates finit qui ont, comme leur nom l'indique, accès à un compteur. À chaque transition, ils peuvent :

— avancer normalement
— incrémenter le compteur,
— le décrémenter,
— tester s'il est à 0 (et choisir le prochaine état en fonction).

Au début, l'automate peut initialiser le compteur.
À la fin, il peut aussi y avoir un test à 0 avant au d'accepter/refuser le mot.

Automates à poids. Les automates à poid sont des automates déterministes avec, sur chaque transition, un entier appelé "poid", et on renvoit à la fin la somme de poids parcourus pour reconnaitre le mot.

Transducteurs Les transducteurs sont des automates finit qui peuvent écrire sur une bande de sortie. cela veut dire qu'à chaque transition, en plus de lire une lettre et de changer d'état, il vont écrire 0, 1 ou plusieurs caractères sur une bande de sortie.
Au début, l'automate peut commencer par écrire quelques caractères initiaux.
À la fin, l'automate peut écrire quelques caractères finaux, en général il n'y a pas d'acceptation/refus du mot.

Automates à pile Les automates à pile sont des automates finit qui ont, comme leur nom l'indique, accès à une pile. À chaque transition, ils peuvent regarder ce qu'il y a en sommet de pile, avant de faire le choix :

— de dépiler ou non le sommet de pile,
— puis d'empiler 0, 1 ou plusieurs caractères,
— puis du nouvel état.

Au début, l'automate peut commencer par initialiser la pile.
À la fin, il peut aussi y avoir un test à vide avant d'accepter ou non le mot.

Automates backtraquants. Les automates backtraquants sont des automates qui peuvent revenir en arrière s'ils se sont trompés ; cela veut dire qu'à chaque transition il peuvent :

— renvoyer une exception pour backtraquer
— tester un premier choix et faire le second en cas d'erreur.

Au début, l'automate peut commencer par tester un choix d'état initial.
À la fin, l'automate peut choisir de backtraquer une fois de plus au lieu d'accepter/refuser.

Automates bidirectionels. Les automates bidirectionels sont capables de changer d'état en avancent ou reculant sur la bande ; cela veut dire qu'à chaque transition il peuvent :

— avancer normalement
— reculer, tout en choisissant le nouvel état.

À la fin, l'automate peut choisir de revenir en arrière au lieu d'accepter/refuser.

2 Les automates monadiques

La structure d'automate monadique est la suivante :

```
type Nom = Int
type TransitionsM mon = Dictinnaire Char (mon Nom)
data EtatM mon = EtatM (mon ()) (TransitionsM mon)
data AutomateFiniM mon = AutomateFiniM (Map Nom (EtatM mon)) (mon Nom)
```

Ce type demande quelques explications :

— `mon` est une monade donnée en paramètre, [1],

1. On ne le dit pas ici, mais il faudra bien mettre (`Monade mon =>`) au début du types de chaque fonctions.

— le `(mon ())` de la définition de `(EtatM mon)` désigne ce qu'il se passe lorsque le mot s'arrête là : au lieu de renvoyer un booléen on renvoi un `(mon ())`, cela veut dire que le résultat est moralement renvoyé à l'environnement monadique directement,
— le "contenu" de `TransitionsM` est de type `(mon Nom)`, cela veut dire que le choix de l'état suivant peut être l'occasion d'une interaction avec l'environnement monadique,
— l'état initial est remplacé par le type `(mon Nom)`, cela veut dire que son choix peut être l'occasion d'une interaction avec l'environnement monadique,

3 Encodage monadique des différentes variantes d'automates

Automates finis Le type des automates finis devient alors :

```
type AutomateFini = AutomateFiniM Maybe
```

On a alors nos automates de la forme : [2]

```
trans1  = insererListe creer [('a',Just 2),('b',Just 1)]
trans2  = insererListe creer [('a',Just 1),('b',Just 2)]
etat1   = EtatM (Just ()) trans1
etat2   = EtatM Nothing   trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (Just 1)
```

Je me doit de préciser quelques points :
— Puisque l'on a remplacé le type `Bool` renvoyé par le type `Maybe ()`, on remplace le booléen `True` par la valeur `Just ()` et le booléen `False` par la valeur `Nothing`.
— On indique bien un échec (`Nothing`) dans le cas où le caractère n'est ni a ni b.

Automates finis non déterministes. Le type des automates finis non déterministes devient alors :

```
type AutomateFiniND = AutomateFiniM []
```

On a alors nos automates de la forme :

```
trans1  = insererListe creer [('a',[1,2]),('b',[1])]
trans2  = insererListe creer [('a',[1]),('b', [])]
etat1   = EtatM [()] trans1
etat2   = EtatM []   trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) [1,2]
```

2. fromList est une fonction qui construit un ARN et y insère une liste de couple (clé, contenu)

Automates à compteur. Le type des automates à compteur devient alors :

```
type AutomateFiniCompteur = AutomateFiniM (State Int)
```

Pour ça il faut importer `Control.Monad.State`.
On a alors nos automates de la forme :

```
trans1  = insererListe creer [('a',return 1),
                              ('b',(testz (return 2) (decr >> return 2)))]

trans2  = insererListe creer [('a', return 1 ),
                              ('b',incr >> return 2)]
etat1   = EtatM decr trans1
etat2   = EtatM incr trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (put 2 >> return 1)

incr :: State Int ()
incr = (\x-> put (x+1)) =<< get
decr :: State Int ()
decr = (\x-> put (x-1)) =<< get

testz :: State Int a -> State Int a -> State Int a
testz x y = do
  n <- get
  if (n==0) then x else y
```

Erratum : Dans le projet de cette année, vous n'arriverez pa a utiliser cette monade, car il faut pour celà une instance de `Eq (State Int Int)`. Un moyen un peu sale est de dire que deux états sont toujours diférents :

```
instance Eq (State a b) where
  x == y   =  False
```

Une solution plus propre serait d'implémenter votre propre version de State, mais ca devient beaucoup trop compliqué.

Automates à poid. Un automate à poid est alors un automate monadique sur le monoid des sentier avec sum :

```
type AutomateFiniPoid = AutomateFiniM (Writer Sum)
```

où (comme vu au TD3) :

```
newtype Sum = Sum Int
```

est un wraper autour des entiers pour préciser que l'on regarde le monoid additif.

On a alors nos automates de la forme :

```haskell
trans1  = insererListe creer [('a',writer (1, 25)),('b',writer (2, 0))]
trans2  = insererListe creer [('a',writer (1, 5)),('b',writer (2, 12))]
etat1   = EtatM (tell 6) trans1
etat2   = EtatM (tell 0)   trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (writer (1, 0))
```

Erratum : Dans le projet de cette année, vous n'arriverez pa a utiliser cette monade, car il faut pour celà une instance de `Eq (Writer Sum Int)`. Pour ca vous pouvez utiliser :

```haskell
instance (Eq a, Eq b) => Eq (Writer a b) where
  x == y   =  runWriter x == runWriter y
```

Transducteurs. Le type des transducteurs devient alors :

```haskell
type Transducteur = AutomateFiniM (Writer String)
```

Pour ça il faut importer `Control.Monad.Writer`.
On a alors nos automates de la forme :

```haskell
trans1  = insererListe creer [('a',(writer (1,"foo "))),('b',(writer (2,"bar ")))]
trans2  = insererListe creer [('a',(writer (1,"hey "))),('b',(writer (2,"oups ")))]
etat1   = EtatM (tell "fin ") trans1
etat2   = EtatM (tell "fin2 ") trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (writer (1,"debut "))
```

Erratum : Voire Erratum sur les automates à poids.

Automates à pile. Le type des automates à pile (sur les caractères comme alphabet de pile) devient alors :

```haskell
type AutomateFiniPile = AutomateFiniM (State String)
```

Pour ça il faut importer `Control.Monad.State`.
On a alors nos automates de la forme :

```haskell
trans1  = insererListe creer [('a',return 1),
                              ('b',(poptest b (return 2) (push c >> return 2)))]

trans2  = insererListe creer [('a', pop >> return 1 ),
                              ('b', push b  >> return 2)]
etat1   = EtatM decr trans1
etat2   = EtatM incr trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (put 2 >> return 1)

push :: Char -> State String ()
push c = (\l-> put (c:l)) =<< get
pop :: State String Char
```

```haskell
pop = state (\(x:l)-> (x,l) )

poptest :: Char -> State String a -> State String a -> State String a
popz c x y = do
  x <- pop
  if (x==c) then x else y
```

Erratum : Voire Erratum sur les automates à compteur.

Automates backtraquants. Le type des automates backtraquants devient alors :

```haskell
type AutomateFiniBack = AutomateFiniM (Cont Bool)
```

Pour ça il faut importer `Control.Monad.Cont`.
On a alors nos automates de la forme :

```haskell
trans1  = insererListe creer [('a',return 1),
                              ('b',try 1 before 2))]

trans2  = insererListe creer [('a', backtrack ),
                              ('b', return 2)]
etat1   = EtatM (return ()) trans1
etat2   = EtatM backtrack   trans2
automat = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) (try 1 before 2)

backtrack :: Cont Bool a
backtrack = cont (\_->False)

tryBefore :: a -> a -> Cont Bool a
tryBefore x y Cont = cont (\k -> k x || k y)
```

Erratum : Voire Erratum sur les automates à compteur.

Automates bidirectionels. Le type des automates bidirectionels devient alors :

```haskell
type AutomateFiniBD = AutomateFiniM (Cont (Either Bool Nom))
```

Pour ça il faut importer `Control.Monad.Cont`.

Erratum : Voire Erratum sur les automates à compteur.

4 Questions

4.1 Questions Simples

Les questions de cette phase sont :

1. Modifier la fonction de reconnaissance pour qu'elle ai le type générique :

   ```
   reconnise :: Monad m => AutomateFiniM m -> String -> m ()
   ```

   deux petites remarques :
   — lorsque l'on cherche une transition mais que l'on ne trouve pas le caractère à lire, on renvoit `fail ""`, ce qui renvoit une erreur dans la monade (Nothing, liste vide, ou une erreur habituel, selon la monade utilisée),
   — il faut souvent post-composer la fonction reconnaître pour avoir le comportement recherché (on va revenir là dessus) ;
2. Modifier `reconiseF` et `figer` avec les mêmes contraintes.
3. Adapter le reste des fonctions pour les versions monadisés (mais sans généricité, voir caneva).

4.2 Questions Avancées (Bonus)

Les questions avancées sont des questions bonus. Ces questions représentaient l'ojectif initial pour ce projet, mais celui-ci a due être revu légèrement à la baisse car trop exigeant.

1. (bonus) Écrivez la fonction de déterminisation monadique décrite dans la section suivante.
2. (bonus) Écrire la fonction `effet` décrite dans la section suivante.
3. (bonus) Écrire la fonction `compM` décrite dans la section suivante.
4. (bonus) Écrire la fonction `starM` décrite dans la section suivante.

5 Les automates non-déterministes monadiques

On a dit qu'un automate non-déterministe était un automate monadique sur la monade des listes. Mais ce n'est pas tout à fait vrais. Mathématiquement, c'est un automate monadique sur la monade des ensembles (les vrais, ceux des mathématiques), que l'on encode par des listes. En effet, considérer qu'il s'agit d'ensembles (sans ordre ni répétition) est absolument essenciel pour l'algorithme de déterminisation.

En fait, il est possible de généraliser l'algorithme de déterminisation pour déterminiser d'autres monades que les ensembles implémentés par les listes. En termes mmathématique, on parle de monades réultant de dustribution sur la monade des ensemble. Mais concrètement, il s'agit d'implémenter la classe suivante :

```
class Monad mon => Determinisable mon where
  distrib :: [mon a] -> mon [a]
  proj    :: [mon a] -> [a]
```

Pour information (celà ne vous sera PAS nécéssaire !), voici les équations que l'on demanderait : [3]

```
  distrib (map return l)  ~= return l
  distrib [u] ~= fmap singleton u
  fmap concat (distrib (distrib l)) ~= distrib (concat l)
  id =<< (fmap distrib (distrib l)) ~= distrib (map (>>= id) l)
```

Un exemple de monade déterminisable est Maybe, vu comme une erreur :

```
instance Determinisable Maybe where
  distrib [] = Just []
  distrib (u:l) = do
     x <- u
     l' <- distrib l
     return (x:l)
```

Celà permet d'écrire un type d'automates monadiques non-déterministes :

```
type NomND = Int
type TransitionsNDM mon = Dictionnaire Char [mon NomND]
data EtatNDM mon = EtatNDM (mon ()) (TransitionsNDM monND)
data AutomateFiniNDM mon = AutomateFiniNDM (Map Nom (EtatNDM mon)) [mon Nom]
```

par exemple, on a l'automate :

```
trans1  = insererListe creer [('a',[Just 2]), ('b',[Nothing])]
trans2  = insererListe creer [('a',[Just 2]), ('b',[Just 1, Just 2])]
etat1   = EtatM [Nothing] trans1
etat2   = EtatM [Just ()]   trans2
automate :: AutomateFiniNDM Poid
automatePoid = AutomateFiniM (fromList [(1,etat1),(2,etat2)])
                             [Just 2]
```

reconnait les mots de la forme $((a+b)a)^*$, mais pas de la manière standard car plutôt que rechercher l'existance d'un chemin reconnaissant le mot dans l'automate, on vérifie aussi l'absence de de chemin relevant une erreur.

On peut alors définir une fonction de déterminisation :

```
determinise :: (Determinisable mon) => AutomateFiniNDM mon -> AutomateFiniM mon
```

De plus, celà permet de généraliser la somme d'automate :

```
(<+>) :: (Determinisable mon) =>
         AutomateFiniNDM mon -> AutomateFiniNDM mon -> AutomateFiniNDM mon
```

3. Ces égalités sont a considérer modulo ordre et duplication sur les listes qui représentent des ensembles.

On peut également généraliser la composition et l'étoile de Kleen :

```
(<.>) :: (Determinisable mon) =>
         AutomateFiniNDM mon -> AutomateFiniNDM mon -> AutomateFiniNDM mon
starM :: (Determinisable mon) =>
         AutomateFiniNDM mon -> AutomateFiniNDM mon
```

Simplement, dans les deux cas, il faut une notion d'état final ; on considère qu'un état est final si est de la forme `EtatM l trans` pour `l` non vide. Par contre, les effets présents dans le `l` ne doivent pas être oubliés et doivent être reportés sur les transitions.

Par exemple si l'on compose l'automate si-dessus avec lui même, on obtient : [4]

```
trans11  = insererListe creer [('a',[Just 12,Nothing]), ('b',[Nothing,Nothing])]
trans12  = insererListe creer [('a',[Just 12,Just 22]), ('b',[Just 11, Just 12,Just 21, Just
trans21  = insererListe creer [('a',[Just 22]), ('b',[Nothing])]
trans22  = insererListe creer [('a',[Just 22]), ('b',[Just 21, Just 22])]
etat11   = EtatM [] trans1
etat12   = EtatM []   trans2
etat21   = EtatM [Nothing] trans1
etat22   = EtatM [Just ()]   trans2
automate :: AutomateFiniNDM Poid
automatePoid = AutomateFiniM (fromList [(11,etat11),(12,etat12),](21,etat21),(22,etat22))
                             [Just 12]
```

On peut aussi rajouter une fonction qui fait un effet à la fin d'un automate non-déterministe "standard" :

```
effet :: AutomateFiniND -> mon () -> AutomateFiniNDM mon
```

Le programme `effect aut after` prend un automate non-déterministe classique `aut::AutomateFiniNDM Identity`, ainsi qu'une précédure `after :: mon()` et qui renvoit l'automate non-déterministe monadique avec :

- les mêmes états que `aut`,
- les mêmes transitions, mais composés avec `return`,
- tous les états finaux renvoient maintenant le rendu final `[after]`,
- tous les états non finaux renvoient maintenant le rendu final `[]`,
- les états initiaux sont pécédés d'un `return`.

Exemple :
Voici le code de `effect pariteA (Nothing)` sur l'automate `pariteA` qui reconnait les mots avec un nombre parie de `'a'` :

```
trans1  = insererListe creer [('a',[return 2]),('b',[return 1])]
trans2  = insererListe creer [('a',[return 1]),('b',[return 2])]
etat1   = EtatM [] trans1
etat2   = EtatM [Nothing] trans2
pariteAEff :: AutomateFiniNDM Poid
pariteAEff = AutomateFiniM (fromList [(1,etat1),(2,etat2)]) [return 1]
```

4. L'automate résultant de cet exemple n'est pas folement intéressant, désolé...

6 MonadeFail [5](Bonus+++)

Les encodage ci-dessus vont parfois crasher : en effet, certaines monades utiliser, comme `State Int`, ne sont pas des monadeFail. Du coup, lorsque l'on essais de reconnaître un mot dont les caractères ne sont pas dans le dictionnaire des transitions, une fonction fail est appelée qui renvoi une exception non rattrapable.

Pour contourner cette erreur, il faut utiliser de vrais monadeFails, quitte à les encoder (car Haskell n'en a pas encore).

```
class Monad m => MonadFail m where
  realFail :: m a
```

On a donc :

```
instance MonadFail Maybe where
  realFail = Nothing

instance MonadFail [] where
  realFail = []
```

avec ça on peut donc utiliser `MonadFail` à la place de `Monad` et `realFail` à la place de `fail`.

Il faut maintenant récupérer les autres variantes.

Celles avec continuation se récupèrent en utilisant une continuation par défaut : [6]

```
class Defaut a where
  defaut :: a

instance Default a => MonadFail Cont a where
  realFail = cont (\_-> defaut)

instance Defaut Bool where
  defaut = False

instance Defaut a => Defaut Either a b where
  defaut = Left defaut
```

Les autres s'obtiennent en enrichissant la monade avec du maybe :

```
instance Monad m => MonadFail MaybeT m where
  realFail = fail
```

5. Cette section est brute de décofrage... Je ne m'atends pas à ce que personne ne se lance dans cette partie, mais il faut la mettre pour être complet. Si vous êtes intéressé et voulez l'essayer, envoyez moi un mail pour demander plus de détails.

6. Il faut ici travailler dans une extension de language : tapez `{-# LANGUAGE FlexibleInstances #-}`

Il faut alors modifier les version qui utilisaient une modade sans échec :

```
type AutomateFiniCompteur = AutomateFiniM (MaybeT (State  Int))
type Transducteur         = AutomateFiniM (MaybeT (Writer String))
type AutomateFiniPile     = AutomateFiniM (MaybeT (State String))
```

Faites tous les changements imposés.