Principes de Programmation
Troisième Phase du Projet de Programmation

29 mars 2019

Cette phase consiste à encoder des automates finis de façon efficace. On a vu en TP un encodage des automates, mais ceux-ci peuvent être infinis ; cela les rend plus puissant mais les privent des avantages qu'ont les automates finis, comme la minimisation.

Il s'agit d'une implémentation beaucoup plus proche de ce qu'on l'utiliserait en programmation impérative. En programmation impérative, un automate est décrit par un tableau bidimentionel de transitions dont la ligne décrit l'état et la colonne décrit le caractère lue. Ici, on ne veut pas utiliser une telle méthode car le tableau serait trop gros (il y a 250000 caractères et beaucoup d'états...) et donc trop lent. On va donc utiliser un dictionnaire plutôt qu'un tableau.

Dans un premier temps, voici des questions préliminaires pour vous familiariser avec le concept. Ces questions ne sont pas obligatoire et ne font même pas partit du projet :

1. Écrivez un automate avec le type suivant : [1]

```haskell
type Nom = String
type Transitions = [(Char,Nom)]                    -- une transition = char+nom_etat
data Etat = Etat Bool Transitions                  -- terminal et transitions
data AutomateFini = AutomateFini [(Nom,Etat)] Nom  -- dico des etats et etat initial
```

2. (non évalué) Écrivez l'automate suivant :

```haskell
etat1 = True  [('a',"q1"),('b',"q2")]
etat2 = False [('a',"q3"),('b',"q1")]
etat3 = False [('a',"q2"),('b',"q1")]
automate = AutomateFini [("q1",etat 1), ("q2",etat2), ("q3",etat3)] 1
```

3. (non évalué) Que fait cet automate ?
4. (non évalué) Écrire un automate qui lit les mots de la forme $(abc)^*$
5. (non évalué) implémentez `reconnait :: AutomateFini -> String -> Bool`.

1. Ici, le type "Nom" est un type ordonné arbitraire, on a utiliser Int, mais vous pouvez utiliser String ou Integer par exemple. Utiliser un alias de type permet de faire ce changement aisément par la suite.

Il faut donc deux types des dictionnaires : celui qui représente les transitions, qui associe un nom à chaque caractère, et celui qui recense les états et qui associe un état à chaque nom.

Ces deux dictionnaires ont des utilisations très différentes et vont être implémentés par des structures différentes : Le premier associe des caractères (énumérables et ordonnés) à des noms (disposant d'une égalité) et a un espace de clé très gros (1114112 caractères) par rapport aux nombres de valeurs (quelques dizaines par dictionnaire), notre implémentation compressée du dictionnaire est donc idéale. Le second associe des noms (ordonnés, non nécessairement énumérables) à des états (qui n'implémentent aucune classe...), avec chaque association visant une valeur unique (un état est pointé par un unique nom), notre version des dictionnaire de fonctionne donc pas, et on utilisera plutôt celle de la librairie standard appelée `Map` [2] qui utilise des AVLs.

Un automate fini sur de l'UTF8 est représenté par le typage suivant. L'idée est que chaque état de l'automate est déterminé par son nom ; pour récupérer un état grâce à son nom, il faut faire une recherche dans le dictionnaire (`Map Nom Etat`) des états. De plus, les transitions sont représentées, non pas comme une fonction, mais comme un dictionnaire (`Dictionnaire Char Nom`) associant un nom d'état à chaque caractère.

```
type Nom = String
type Transitions = Dictionaire Char Nom
data Etat = Etat Bool Transitions
data AutomateFini = AutomateFini (Map Nom Etat) Nom
```

Par exemple, voici l'automate trivial qui reconnaît le mot vide :

```
e0 :: Etat
e0 = Etat True creer

autVide :: Automate
autVide = Automate (fromList [("etat0", e0)]) "etat0"
```

Un autre exemple est l'automate acceptant un nombre paire de 'a' :

```
e1 :: Etat
e1 = Etat True (insererIntervalle (inserer creer 'a' "etat2'') 'b' 'z' "etat1")
e2 :: Etat
e2 = Etat False (insererIntervalle (inserer creer 'a' "etat1'') 'b' 'z' "etat2")

pariteA :: Automate
pariteA = Automate (fromList [("etat1", e1),("etat2", e2)]) "etat1"
```

Vous allez devoir :

1. Écrire la reconnaissance/évaluation

   ```
   reconise :: AutomateFini -> String -> Bool
   ```

2. À ne pas confondre avec la fonction `map` qui n'a rien à voire (homonyme malheureux...).

(attention au cas où le caractère lu n'est pas dans l'arbre des transitions)
qui fonctionne comme suit :
on part avec le nom i de l'état initial et un mot w_0 ;
on cherche l'état e_0 correspondant à i dans le dictionnaire ($etats$ `:: Map Nom Etat`) ;
on échoue si on ne le trouve pas ;
si $w_0 = \epsilon$ est le mot vide, on renvoi le booléen de terminaison de e_0 ;
sinon on récupère le premier caractère c_0 du mot $w_0 = c_0 w_1$;
on récupère le dictionnaire (t_0 `:: Dictionnaire Char Nom`) de transition de l'état initial e_0 ;
on utilise c_0 pour récupérer le nom n_1 du prochain état ;
on cherche l'état e_1 correspondant à n_1 dans t_0 `:: Dictionnaire Char Nom` ;
on échoue si on ne le trouve pas ;
si $w_1 = \epsilon$ est le mot vide, on renvoi le booléen de terminaison de e_1 ;
sinon on récupère le premier caractère c_1 du mot $w_1 = c_1 w_2$;
on récupère le dictionnaire (t_1 `:: Dictionnaire Char Nom`) de transition de l'état e_1 ;
on utilise c_1 pour récupérer le nom n_2 du prochain état ;
etc...

Par exemple, `reconise pariteA "sdedfzeazdaadza"` doit rendre `True`, mais `reconise pariteA "sdedfzeazdadza"` doit rendre `False`, ainsi que `reconise pariteA "a0a"`, car dans le dernier cas, on prend une transition non définit.

2. Écrire une fonction

```
newAutomaton :: [(Nom,Bool)] -> [(Nom,Char,Nom)] -> AutomateFini
```

qui, à une liste de noms et de transitions associe l'automate.
Par exemple, une version de `pariteA'` n'acceptant que des mots de 'a' et de 'b' peut s'écrire :

```
pariteA' = newAutomaton [("etat1",True),("etat2",False)]
                        [("etat1",'a',"etat2"),
                         ("etat1",'b',"etat1"),
                         ("etat2",'a',"etat1"),
                         ("etat2",'b',"etat2")]
```

3. Écrire une fonction

```
forgetFinitness :: AutomateFini -> AutomateD
```

qui transporte un automate fini en un automate déterministe du type utilisé dans le TP2,

4. En vous inspirant de la correction du TP2, écrire une version non déterministe `AutomateFiniND` du type des automates finis ; on doit pouvoir représenter plusieurs transitions depuis un même état et sur la lecture du même caractère, on doit aussi pouvoir mettre plusieurs états initiales, on peut aussi autoriser les ϵ-transitions (pas obligatoire).

```haskell
type NomND = String
-- type TransitionsND = ADefinir
data EtatND = ADefinir2
data AutomateFiniND = ADefinir3
```

5. Pour écrire la déterminisation efficacement, vous allez avoir besoin d'opération plus avancé sur les dictionnaires. Voici quatre fonctions sur les dictionnaires écrites dans le cadre de la correction (avec le datatype utilisé),

```haskell
data Dictionnaire cle val =
    E
  | A (Dictionnaire cle val) (cle,cle) val (Dictionnaire cle val)

-- un fold qui peut utiliser les intervalles de clefs
foldIntervalle :: (Ord cle, Enum cle) =>
           (a -> cle -> cle -> b -> a) -> a -> Dictionnaire cle b -> a
foldIntervalle _ acc F = acc
foldIntervalle f acc (A g (l,u) v d) =
   foldIntervalle f (foldIntervalle f (f acc l u v) g) d

-- recompresse un dictionnaire mal compresse
compresser :: (Ord cle, Enum cle, Eq a) =>
              Dictionnaire cle a -> Dictionnaire cle a
compresser = foldIntervalle insererIntervalle creer

-- Cette fonction n'est pas safe : le dictionnaire resultant n'est pas compresse
insererIntervalleCons :: (Ord cle, Enum cle) =>
                      Dictionnaire cle [val] ->
                      (cle,cle)   ->   val   ->   Dictionnaire cle [val]
insererIntervalleCons E intervalle v = A E intervalle [v] E
insererIntervalleCons (A g (l,u) v d) (x,y) w
   | y < l
       =  A (insererIntervalleCons g (x,y) w) (l,u) v d
   | x > u
       = A g (l,u) v (insererIntervalleCons d (x,y) w)
   | x > l && y < u
       = A (A g (l,pred x) v E) (x,y) (w:v) (A E (succ y,u) v d)
   | x == l && y < u
       = A g (x,y) (w:v) (A E (succ y,u) v d)
   | x > l && y == u
       = A (A g (l,pred x) v E) (x,u) (w:v) d
   | x == l && y == u
       = A g (l,u) (w:v) d
   | x == l && y > u
       = A g (l,u) (w:v) (insererIntervalleCons d (succ u,y) w)
   | x < l && y == u
```

```
        = A (insererIntervalleCons g (x,pred l) w) (l,u) (w:v) d
  | x < l && y > u
        = A (insererIntervalleCons g (x,pred l) w)
            (l,u) (w:v)
            (insererIntervalleCons d (succ u,y) w)
  | x < l && y < u
        = A (insererIntervalleCons g (x,pred l) w)
            (l,y) (w:v)
            (A E (succ y,u) v d)
  | x > l && y > u
        =  A (A g (l,pred x) v E)
             (x,u) (w:v)
             (insererIntervalleCons d (succ u,y) w)

-- fait l'union de dictionnaire en recuperant
-- la liste des valeurs en cas de conflit
unionCons :: (Ord cle, Enum cle, Eq val) =>
             [Dictionnaire cle val] -> Dictionnaire cle [val]
unionCons l = compresser (Prelude.foldl unionConsBin creer l)
    where
    unionConsBin d1 E = d1
    unionConsBin d1 (A g (l,u) v d) =
         unionConsBin (unionConsBin (insererIntervalleCons d1 (l,u) v) g) d
```

Ajoutez ces fonctions à la librairie de dictionnaire choisie en adaptant à votre datatype. Rendez `unionCons` visible en l'ajoutant à l'entête de module. La fonction `unionCons` va associer à une clé la liste de toutes les valeurs associés à cette clé dans chacun des dictionnaires donnés.

6. On va avoir besoin d'une autre fonction supplémentaire dans dictionnaire :

   ```
   toListVal :: Dictionnaire cle val -> [val]
   ```

   qui rend la liste des valeurs trouvés dans le dictionnaire (il peut y avoir des duplications, mais il ne faut pas qu'elle soit plus grande que la taille en mémoire du dictionnaire).

7. (non évalué) Écrire une fonction qui a une liste de noms d'états non-déterministes associe le nom de l'état déterminiser correspondant :

   ```
   nommer :: [NomND] -> Nom
   ```

   qui va mettre tous les noms de la liste dans un ensemble pour éviter les répétitions et les permutations, puis transformer cet ensemble en string avec show. On pourrait utiliser nos ensembles, mais on ne peut pas stocker de string dedans car ce ne sont pas des énumérations. On va donc utiliser les `Set` de la librairie Data.Set
   Par exemple, `nommer ["x","toto","e1","toto"]` doit renvoyer quelque chose comme `"[e1,toto,x]"`, ou encore `"fromList [\"e1\",\"toto\",\"x\"]"` qui est le résultat de `show` sur le Set contenant `"x"`, `"toto"` et `"e1"`.

8. (non évalué) Écrire la fonction

   ```
   estTerminalList :: [EtatND] -> Bool
   ```

9. (non évalué) Écrire la fonction

   ```
   regroupe :: [TransND] -> TransND
   ```

   en utilisant notre `unionCons`, ainsi que le `mapDico`.

10. (non évalué) Écrire la fonction

    ```
    fusionT :: [EtatND] -> TransND
    ```

    qui prend une liste d'états non déterministes, en extrait la liste des transitions et les regroupes.

11. (non évalué) Écrire la fonction

    ```
    creerEtat :: [EtatND] -> Etat
    ```

    qui crée l'état déterministe associé à une liste d'états non déterministes. Il est final si l'un des états non déterministes l'étaient. Les transitions sont obtenus en appliquant `nommer` sur toutes les valeurs obtenues après application de `fusionT`.

12. (non évalué) Écrire la fonction

    ```
    deterministeAux :: (Map NomND EtatND) -> Map Nom Etat -> [[NomND]] -> Map Nom Etat
    ```

    qui prend une association nom-etat non déterministe (celui de l'automate non déterministe) `etatsND`, une association nom-etat déterministe (il s'agit d'un accumulateur) `acc` et une liste de listes d'états non déterministes `l` et agit ainsi :
    - si `l` est vide, on renvoi l'accumulateur
    - si `l=x:l'` et que la déterminisation du nom de `x` est une clé de `acc`, on récurse simplement sur `l'`,
    - sinon, on récupère la liste `es` des états non déterministes dont le nom est dans `x`, on l'utilise pour créer un nouvel état avec `creerEtat`, que l'on insère dans l'accumulateur (avec déterminisation du nom de `x` comme clé), on va alors récurser avec ce nouvel accumulateur, mais il faut ajouter de nouveaux états à la liste `l` qui sont les états accessibles depuis l'état nouvellement créé. Pour calculer ces états, il faut prendre la fusion des transitions de `es` et en extraire toutes les valeurs.

13. (non évalué) Dans un second temps, essayez de modifier ce code pour que la déterminisation du nom de `x` ne soit calculer qu'une fois, de même que la fonction `fusionT es`

14. Écrire une fonction de déterminisation

    ```
    determinise :: AutomateFiniND -> AutomateFini
    ```

    qui calcule le déterminiser en prenant la déterminisation du nom des états initiaux comme état initial, et calcule les états en appelant `deterministeAux` sur l'accumulateur vide et la liste avec l'ensemble des états initiaux comme seul élément.

15. (bonus) Essayer de tout faire avec des `Set` plutôt que des listes.
16. Écrire une fonction

    ```
    segment :: Char -> Char -> AutomateFiniND
    ```

    qui prend deux caractère et rend un état qui reconnait le langage des mots d'un caractère c compris entre ces deux caractères entrés. Par exemple, `segment 'a' 'c'` reconnait le langage `{"a","b","c"}`. Pour les noms d'états, utilisez les caractères entrés.
17. En vous inspirant de la correction du TP2, écrire une fonction de somme d'automates

    ```
    (!+!) :: AutomateFiniND -> AutomateFiniND -> AutomateFiniND
    ```

    on doit avoir que la somme de A_1 et A_2 reconnait w si w est reconnu par A ou si w est reconnu par B. Par exemple, `(segment 'a' 'b')!+! (segment 'x' 'x')` reconnait le langage `{"a","b","x"}`. L'idée est de modifier le noms des états des automates pour qu'il n'y ai pas de conflit de nom entre les deux et de prendre l'union des états.
18. En vous inspirant de la correction du TP2, écrire une fonction d'étoile de Kleen

    ```
    star :: AutomateFiniND -> AutomateFiniND
    ```

    on doit avoir que l'étoilé de A reconnait les mots de la forme $w_1...w_n$ où w_i est reconnu par A pour tout i. Par exemple, `star (segment 'a' 'b')` reconnait le langage `{"","a","aa","aaa","aaaa",...}`. L'idée est d'ajouter des transitions (non déterministes) depuis les états finaux de A vers les cibles de ses états initiaux.
19. En vous inspirant de la correction du TP2, écrire une fonction de concaténation

    ```
    (!.!) :: AutomateFiniND -> AutomateFiniND -> AutomateFiniND
    ```

    on doit avoir que la concaténation de A_1 et A_2 reconnait les mots de la forme w_1w_2 où w_1 est reconnu par A et w_2 est reconnu par B. Par exemple, `(segment 'a' 'b')!.! (segment 'x' 'y')` reconnait le langage `{"ax","bx","ay","by"}`. L'idée est de modifier le noms des états des automates pour qu'il n'y ai pas de conflit de nom entre les deux, de prendre l'union des états, puis d'ajouter des transitions (non déterministes) depuis les états finaux du premier automate vers les cibles des états initiaux du second, et enfin de rendre non-finaux les états finaux du premier automate.
20. Récupérez le datatype pour les expressions régulières :

    ```
    data Regexp =  Segment Char Char
                 | Regexp :+: Regexp
                 | Star Regexp
                 | Regexp :.: Regexp
    ```

et écrire une fonction qui associe à une expression régulière son automate.

```
regexp2Automata :: Regexp -> AutomateFini
```

21. (bonus) En utilisant l'algorithme de votre choix, écrire une fonction de minimisation de l'automate

    ```
    minimisation :: AutomateFini -> AutomateFini
    ```

    Ce programme ne sera pas testé automatiquement, juste examiné, car on ne vous demande pas d'être efficace, ce qui serait très difficile étant donné que l'ensemble des caractères, `[minBound..maxBound] :: [Char]` est très gros.

Attention : Même si vous ne touchez par aux modules Ensemble et des Dictionnaire (vous avez le droit d'y toucher ;), veuillez les joindre à l'archive final !