Principes de Programmation
Quatrième Phase du Projet de Programmation

8 avril 2019

Dans la phase précédente, le choix des noms des états pendant la construction était un casse tête, ne semblais pas sure (on risque les conflits de noms) et créer des noms énorme qui sont coûteux à manipuler. Dans cette phase, on essais de rendre l'utilisation des noms plus efficace.

Hashage des dictionnaires et Set Instanciez `Hashable Set`, de sorte à ce que deux ensemble de même contenu aient le même hash, mais des ensembles aux contenus différent aient (a priori) des hash différents. Dans le canevas, on utilise `hash s = hash (toList s)`, mais vous pouvez en changer.

Instanciez `Hashable Dictionnaire` de sorte à ce que deux dictionnaires associant les mêmes clés aux mêmes valeurs aient le même hash, mais des dictionnaires aux contenus différent aient (a priori) des hash différents. Attention à rester efficace et à ne pas parcourir tous les éléments contenus dans un segment...

Des Int comme noms Comparer des string n'est pas efficace du tout, changez le type de `Nom` pour des entiers et utilisez une fonction de hashage pour créer de nouveaux noms chaque fois que nécessaire :

- pour la déterminisation, utiliser le hash des ensemble décrit plus haut pour obtenir le nouveau nom
- pour le renomage disjoint pour la concaténation, l'idée est d'utiliser un sel différent pour réencoder les noms de l'un et l'autre.
- pour la fonction segment : prenez le hash du couple des bornes du segment...

Des automates figés Lors de la reconnaissance, à chaque transition, récupérer le nom d'un état puis aller chercher l'état en question n'est vraiment pas efficace. On voudrait avoir directement accès à l'état. Pour ça il faut changer le type d'automate figé, on dit qu'il est figé, car on ne peux pas le construire directement, mais uniquement en passant par un automate déterministe :

```
type TransitionsF = Dictionnaire Char EtatF
data EtatF = EtatF Bool TransitionsF
type AutomateFiniF = EtatF
```

```
figer :: AutomateFini -> AutomateFiniF
reconiseF :: AutomateFiniF -> String -> Bool
```

Minimisation efficace Les algorithmes standard de minimisation ne sont pas efficace lorsque l'alphabet est trop gros (ils sont linéaires en la taille de l'alphabet). Il faut donc utiliser d'autres algorithme. On se propose ici d'utiliser un algorithme maison utilisant les hashs.

- Comme dans les algorithmes standards, on maintient une table de partition des états. Celle-ci prend la forme d'un dictionnaire (ou d'un Map)

  ```
  type Partition = Map Nom Int
  ```

  associant les noms de nos états à un entier appelé classe d'équivalence (deux états étant dans le même classe d'équivalence s'ils sont associés au même entier).
- On utilise une fonction

  ```
  taillePartition :: Partition -> Int
  ```

  qui compte combien de valeurs différentes on a (càd. combien de classe on a dans la partition), que l'on encodera.
- On utilise une fonction

  ```
  projection :: Partition -> Transition -> Transition
  ```

  qui prend une table de partition, et une transition et qui compose les deux de sorte à associer à chaque caractère le nom de la classe d'équivalence auquel appartient l'état ciblé par la transition.
- Au début, on associe le nombre 0 aux états non terminaux et 1 aux états terminaux.

  ```
  partitionInnitial :: AutomateFini -> Partition
  ```
- Au début de chaque étape, on mesure la taille de la partition, puis on calcule une nouvelle partition

  ```
  nouvellePartition :: AutomateFini -> Partition -> Partition
  ```

  pour ça on passe sur chaque état e :
  - on projette le dictionnaire de transitions le long de la table des partitions,
  - puis on fait le hash du résultat, il s'agit du nom de la classe d'équivalence associé à l'état traité dans la nouvelle partition.
- À la fin on mesure la nouvelle table de partition, si elle est plus grosse que l'ancienne, on relance l'algorithme sur cette nouvelle table.
- Une fois que l'on arrive à un point fixe, on dispose d'une table de partition optimal. On crée un nouvel automate

  ```
  partition2Automate :: AutomateFini -> Partition -> AutomateFini
  ```

  dont :
  - les nom des états sont les noms des différentes classes d'équivalence,

— pour chaque classe d'équivalence, on crée un état dont les transitions sont le projeté du dictionnaire de transitions d'un des états de la classe (peu importe lequel) le long de la table des partitions.

Bonus : Sur un mot très grand, un automate peut être lancé en parallèle, mais ce n'est pas simple du tout. Un grand mot est une liste de string :

```
type BigString = [String]
```

Le mot total est la concaténation du grand mot, mais on peut y accéder en parallèle.

Pour lancer un automate en parallèle sur un grand mot, il faut calculer un dictionnaire qui à chaque nom d'état de l'automate renvoi 0 ou un autre nom d'état :

```
executAll :: AutomateFini -> String -> Map Nom Nom
```

l'idée étant que si on change l'état initial de `aut` par `e`, et si en évaluant `w` sur l'automate on finit dans l'état `e'` (final ou non), alors `(toDico aut w) ! e == e'`, et si on échoue avant alors `(toDico aut w) ! e == 0`. Par exemple sur l'automate `paritA` a deux états calculant la parité de 'a', on aurait

```
toDico paritA "bbaaab" == fromList [(1,0),(0,1)]
```

qui échange les états.

Une fois que l'on a cette fonction, on peut utiliser le `parMap` de la librairie `Control.Parallel.Strategies` pour appliquer cette fonction à tous les petits mots du grand mot, on utilise pour ca la stratégie donnée `evalDico`, puis on calcule l'état initial en parcourant la liste.

Attention, cette nouvelle fonction de reconnaissance n'est vraiment efficace que si on a accès à plus de coeur de calcul qu'il n'y a d'état dans l'automate... (en pratique un peu mieux mais pas beaucoup)