Automates et applications

Paul Patault

14 mars 2022

Résumé

Projet du cours *Automates et Applications*. Utilisation de la structure de donnée d'automates d'arbres pour vérifier la bonne formation de fichier type XML. Ces automates d'arbres seront générés automatiquement à partir d'un fichier *à la* DTD donné en entrée du programme. Celui-ci sera donc parsé puis compilé dans notre type.

1 Validation des documents

Section sans questions. Nous profitons cependant de cet espace pour faire quelques remarques.

1.1 Syntaxe

1.1.1 Fichiers DTD

L'extension `.dtd` est utilisé comme abus de langage mais clarifie les choses. La syntaxe est simple, un type commence par le mot clé « `type` » suivi par le nom donné à ce dernier, ensuite un signe « `=` » puis le nom de la balise et enfin entre « `[]` » se trouve une expression régulière (où « `*` » est l'étoile de Kleene, « `|` » indique l'alternative et « `+` » la concaténation). Voici un fichier `.dtd` qui sera accepté par le parser :

```
type t  = HD[ x* ]
type x  = P[ y+z* ]
type y  = B[ (z+z)|x* ]
type z  = C[]
```

1.1.2 Fichiers XML

Nous utilisons la bibliothèque `Xml-light` pour le parsing. Voici un fichier `.xml` d'exemple qui sera accepté par le parser (remarque : ce fichier est bien typé par rapport au fichier `.dtd` proposé en exemple) :

```
<HD>
    <P>
       <B>
          <C></C>
          <C></C>
       </B>
    </P>
    <P>
       <B></B>
       <C></C>
       <C></C>
    </P>
</HD>
```

1.2 À propos du sujet, et de l'implémentation

Je n'ai pas réussi à faire le sujet complet. Le pseudo-code de la section 4 n'est pas exact, et le code réel associé ne mène pas encore à un résultat concluant. Je n'ai pas complètement trouvé comment concevoir l'algorithme de compilation d'un fichier de type `DTD` vers un automate d'arbre binaire[1]. Le point qui m'a en particulier posé problème est la compilation des transitions. En effet, je ne trouve pas de façon générale pour réaliser cette transformation. Je crois que mon problème est précisément situé au niveau des états à nommer avant la vérification de cohérence. Ayant un code pour gérer les expressions régulières, il faudrait « simplement » transporter en plus l'état courant de l'automate classique en plus.

1. Je précise bien binaire car il s'agit selon moi (je peux me tromper) d'une difficulté supplémentaire imposée dans ce projet, je pense qu'avec une structure d'arbre classique la compilation des transitions aurait été plus simple.

2 Validation top-down non-déterministe

2.1 Question

Un run d'un automate d'arbre $A = (Q, \delta, I, F, \Sigma)$ pour un arbre $t \in \mathcal{T}(\Sigma)$ est une fonction $r : dom(t) \to Q$ telle que $\forall p \in dom(t), (t(p), r(p), r(p1), r(p2)) \in \delta$. Un run est dit acceptant si et seulement si $r(\epsilon) \in I$.

2.2 Question

Algorithm 1 Pseudo-code *à la* Caml pour la fonction `validate_td`

```
let rec validate_td a t p q =
  if label (t p) = '#' then is_final q
  else
    let l =
      List.filter
        (fun (l, q', _, _) -> l = label (t p) && q = q)
        a.delta
    in
    List.fold
      (fun acc q' ->
        acc
        || validate_td a t (p @ [ first_child (t p) ]) q'
           && validate_td a t (p @ [ next_sibling (t p) ]) q')
      false l
```

2.3 Question

La complexité de l'expression $\exists$ `q` $\in I$ tel que `validate_td a t eps q` est $O(|\mathtt{a}|^{|\mathtt{t}|})$, où $|\mathtt{a}|$ est le nombre de transitions de l'automate `a`. En effet, l'algorithme nous fait prendre au pire $|\mathtt{a}|$ fois chaque arête de l'arbre `t`.

3 Validation bottom-up

3.1 Question

Algorithm 2 Pseudo-code *à la* Caml pour la fonction `validate_bu`

```
let rec validate_bu a t p =
  let lab = label (t p) in
  if lab = '#' then
    List.filter
      (fun (lab', _, l', r') -> '#',[],[] = lab',l',r' )
      a.delta
else
  let left = validate_bu a t (p@[first_child (t p)]) in
  let right = validate_bu a t (p@[next_sibling (t p)]) in
  let res = ref [] in
  List.iter (fun r -> List.iter (fun l ->
      let trans = lab, l, r in
      let possible_states =
        List.filter
          (fun (lab', _, l', r') -> lab,l,r = lab',l',r' )
          a.delta
      in
      res <- possible_states :: !res;
    ) left) right
    res
```

3.2 Question

La complexité de l'expression (`validate_bu a t eps`) $\cap$ I est $O(|\mathtt{t}|)$.

3.3 Question

TODO

4 Compilation

4.1 Question

En quelques mots, nous commençons par transformer l'arbre n-aire parsé par la librairie `Xml-light` en un arbre binaire (le code est situé dans le fichier `src/tree.ml`) avec un algorithme simple s'appuyant sur l'isomorphisme de ces deux structures présenté en fin de cours 3. Ainsi les fils directs de chaque nœud deviennent les fils droits du fils gauche de ce nœud et récursivement.

Algorithm 3 Pseudo-code *à la* Caml pour la fonction de compilation des arbres n-aires vers des arbres binaires

```
let n2bin input_n_tree =
  let rec aux = function
    | [] -> Leaf
    | node :: sibling -> Node (aux node, aux sibling)
  in
  match input_n_tree with
  | rac, childs -> Node (rac, aux childs, Leaf)
```

Pour la compilation des expressions régulières, nous utilisons la construction trouvée sous le nom de Berry-Sethi [2]. Celle-ci est efficace, et n'a pas le défaut de Thompson en générant des transitions epsilon qu'il aurait par la suite fallut supprimer. Le code se trouve dans le fichier `src/regautomata.ml` et parle de lui-même, nous n'allons donc pas le présenter plus en détails.

Enfin pour la compilation des types *à la* DTD en eux même, la solution n'est pas encore très claire. Arrivé seul à la même idée que celle proposé dans le message sur e-campus [3], l'avancement s'est fait à tâtons et aucune conclusion n'est envisageable dans l'immédiat.

L'implémentation est donc une tentative d'application d'une idée vague relativement similaire à celle discutée. La fonction de compilation d'un fichier de type est située dans `src/compiler.ml` et il s'agit de la fonction `compile_typ`. L'idée est dans un premier temps de remplir un table de hashage contenant les types, pour regrouper les possibles multiples définitions d'un même type. Ensuite, en itérant sur chacun de ces derniers, l'automate de mot correspondant à

l'expression régulière est fabriqué, et nous tentons d'encoder ces états dans les transitions de notre automate d'arbre en construction. Cependant, il doit y avoir une erreur à ce niveau dans le code, mais la localisation précise ce bogue pose encore problème.

4.2 Question

Ayant une complexité en $O(n^2)$ pour la construction de Glushkov, et en entrée n définitions, nous avons une complexité en temps en $O(n^3)$ [4]. Pour la taille de l'automate d'arbre produit correspondant au type donné en entrée, celle-ci serait $O(TODO)$.

5 Implémentation

Le code se trouve dès la racine de ce répertoire, les fichiers sources sont dans le dossier `src/` et le *main* est le fichier `validate.ml` se trouvant lui dans le dossier `bin/`. L'architecture globale est classique, respectant notament les informations proposées par la documentation du builder `dune`. Des explications pour utiliser ce code se trouvent dans le fichier `README.md` à la racine du projet.

Enfin, presque tout a été codé, mais du à un bogue dans la compilation des types, rien n'est complétement fonctionnel.

3. Celle-ci semble être la même que celle de Glushkov.

3. Message que je n'ai pu découvrir que ce dimanche, n'ayant pas eu de notification relative aux messages, probablement dû à leur caractère « semi-privés » sur e-campus.

4. En supposant que la taille des expressions régulières est du même ordre de grandeur que le nombre de types.