université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

À rendre pour le 14 mars 2022

Automates et Applications
M1–MPRI Orsay

Devoir

Modalités

Ce devoir est individuel. Il est à rendre sous la forme d'une archive `.tar.gz` contenant les réponses aux questions et le code source. Les questions $\star$ sont des questions à rédiger dans un fichier (au format PDF, ce dernier peut être généré à partir de Latex, Word, *etc.* mais le fichier PDF final doit se trouver dans l'archive).

Le code source peut utiliser au choix le langage OCaml, Java ou Python. Des exemples de code sont donnés.

1 Validation de documents

Le but de ce devoir est double. Dans un premier temps, on souhaite pouvoir définir des types, semblables aux DTDs pour des documents XML. Les types ont la syntaxe suivante :

```
type t = A[ x* | y ]
type x = B[]
type y = choco[]
```

Formellement, un *ensemble de définitions de types* est une suite de définitions de la forme `type` t `=` l`[` r `]` où t est le nom du type et r est une expression régulière formée sur les noms de types et pouvant contenir les opérateurs « * », « ? » et l'opérateur binaire « | ». Des parenthèses peuvent être utilisées pour désambiguer l'application de l'union. Pour les étiquettes des arbres, représentées par l ce dernières peuvents êtres de la forme :

- Une suite de lettres, _ ou chiffres, commençant par une lettre comme `choco`, pour indiquer une balise.
- Le caractère $*$ pour indiquer que n'importe quelle balise est acceptée.
- L'expression $\sim l_1 \sim l_2 \ldots \sim l_n$ pour indiquer que toutes les étiquettes sauf $l_1, \ldots, l_n$ sont acceptées.

Le but est de compiler de tels types vers des automates d'arbres. On donnera ensuite un programme `validate` pouvant être appelé avec les paramètres suivants [1] :

```
$ validate fichier.typ fichier.xml t
```

On demande ici de valider le fichier `fichier.xml` avec les définitions `fichier.typ` en utilisant `t` comme définition de type pour la racine.

Les fichiers XML seront considérés comme des arbres. On pourra supposer que ces fichiers sont sans aucun caractère blancs, sans attribut, commentaires ou autre élément syntaxique que des balises ouvrantes ou fermantes. Par exemple :

```
<A><B></B><B></B><B></B></A>
```

Il n'y a pas de restriction sur les types comme il pourrait y en avoir dans les DTDs. Ainsi, un fichier de type comme celui-ci est parfaitement valide :

```
type t = f[ x y ]
type t = f[ y x ]
type x = a[]
type y = b[]
```

1. Un programme Java pourra être évidemment appelé avec `java validate fichier.typ fichier.xml t`.

On notera que ce fichier contient deux définitions pour `t`, et que ce type décrit un langage non déterministe descendant. Un autre exemple de type est celui-ci :

```
type lst = L[ any lst ]
type lst = N[]
type any = *[ any* ]
```

Ce dernier représente des « listes chaînées » composées de deux cas, la liste à deux éléments et la liste vide. Les éléments de cette liste peuvent être n'importe quel arbre. Le document XML ci-dessous est par exemple valide pour le type `lst` :

```
<L><hello></hello><L><how><are><you></you></are></how><N></N></L></L>
```

Les documents XML n'ayant pas d'alphabets fixés, on choisit de les encoder par des arbres binaires avec le codage « premier fils, frère suivant ». On supposera un alphabet Σ, dénombrable, de toutes les étiquettes XML possible, considérées comme des symboles binaires ainsi qu'un unique symbole constant $\#$ représentant les feuilles.

Dans ce contexte, un automate d'arbre est de la forme (Q, δ, I, F) où Q est un ensemble d'états, I est un ensemble d'états valides à la racine de l'arbre, F est un ensemble d'états valides aux feuilles et $\delta \subseteq \mathcal{P}_f(\Sigma) \cup \mathcal{P}_{cof}(\Sigma) \times Q \times Q \times Q$ est une *relation* de transition.

Un élément (L, q, q_1, q_2) de la relation signifie que si le sous-arbre courant possède une étiquette $l \in L$ et que l'automate est dans l'état q, alors il doit être reconnu par q_1 sur son sous-fils gauche et q_2 sur son sous-fils droit.

Ici, L est un sous-ensemble fini ou *co-fini* de Σ. On rappelle qu'un ensemble est co-fini si son complémentaire est fini. Le fait que δ soit une relation et non pas une fonction permet d'exprimer le non-déterminisme. Un exemple de tel automate est : $\mathcal{A} = (\{q_0, q_1, q_\perp\}, \delta, \{q_0\}, \{q_0, q_1\})$

$$\delta : \begin{array}{lcl} \{f\}, q_0 & : & q_1, q_0 \\ \Sigma \setminus \{f\}, q_0 & : & q_\perp, q_\perp \\ \Sigma, q_1 & : & q_\perp, q_\perp \\ \Sigma, q_\perp & : & q_\perp, q_\perp \end{array}$$

qui reconnaît les peignes à droite dont l'étiquette est f (et les feuilles $\#$). On note que Σ est co-fini, car son complémentaire est $\varnothing$, qui est fini. On remarque enfin qu'un tel automate peut être vu comme descendant (en lisant les transitions de gauche à droite) ou ascendant (en lisant les transitions de droite à gauche).

2 Validation top-down non-déterministe (5 points)

1. $\star$ (1 point) Donner la définition d'un *run* acceptant adapté au modèle d'automate considéré.
2. $\star$ (2 points) Donner le pseudo-code d'une fonction récursive `validate_td` prenant en argument un automate $\mathcal{A}$, arbre t, un chemin p dans cet arbre et *un* état q de l'automate et renvoyant vrai si le sous arbre enraciné en p est reconnu dans l'état q, et faux sinon.

   On pourra utiliser les fonctions auxiliaires `label`(p), `first-child`(p) et `next-sibling`(p) pour accéder respectivement à l'étiquette du nœud p, à son premier fils ou à son second fils.

   Attention, on rappelle que l'automate est non-déterministe et que plusieurs transitions peuvent être valides pour l'état q et l'étiquette l du nœud courant.
3. $\star$ (2 points) Donner la complexité de l'expression :

   $$\exists q \in I \text{ tel que } \texttt{validate\_td}(\mathcal{A}, t, \epsilon, q)$$

   en fonction de la taille de l'automate considéré et de l'arbre t. On considère que la taille d'un automate est le nombre de transitions.

3 Validation « bottom-up » (5 points)

La procédure donnée dans la section précédente implémente naïvement un automate top-down non déterministe qui essaye tous les choix possibles. On souhaite améliorer la complexité de la validation en procédant par étapes.

1. $\star$ (2 points) Donner le pseudo-code d'une fonction récursive `validate_bu` prenant en argument un automate $\mathcal{A}$, un arbre t, un chemin p dans t et renvoyant un *ensemble d'états* R qui acceptent le sous-arbre p.

   Indication : on souhaite ici simuler une évaluation *bottom-up* l'automate. On devra donc calculer d'abord récursivement l'ensemble R_1 des états acceptés par le sous-fils gauche, puis R_2 celui accepté par le sous-fils droit pour enfin calcule R en fonction de $t(p)$, R_1 et R_2.

2. $\star$ (1 point) Donner la complexité de l'expression :

$$\texttt{validate\_bu}(\mathcal{A}, t, \epsilon) \cap I$$

   en fonction de la taille de l'automate considéré et de l'arbre t.

3. $\star$ (2 points) Une observation que l'on peut faire est que si I est vide alors la procédure `validate_td` s'arrête à la racine alors que la procédure `validate_bu` doit parcourir tout l'arbre pour se rendre compte au final qu'il n'y aucun état intial. Cette observation peut se généraliser récursivement. Proposer une fonction récursive `validate_opt` prenant en argument un automate $\mathcal{A}$, un arbre t, un chemin p dans t, *ensemble d'états candidats* P et qui renvoie le sous-ensemble $R \subseteq P$ d'états acceptant le sous-arbre p. Que doit faire cette fonction si P est vide ou R est vide ?

4 Compilation (4 points)

1. $\star$ (3 points) Donner une procédure (pseudo-code + explications éventuelles) permettant de transformer une définition de types en automate d'arbres. On détaillera en particulier :
   — La façon dont les arbres XML peuvent être vus comme des arbres binaires
   — La façon de compiler les types, et en particulier les expressions régulières dans notre format d'automates

2. $\star$ (1 points) Quelle est la taille de l'automate produit, en fonction de celle du type ? (vous préciserez comment vous calculez la taille du type). En déduire la complexité de tout le processus de validation de document avec notre approche.

5 Implémentation (6 points)

Coder la compilation de types, le chargement et la validation de documents, dans le langage de votre choix.

Le code devra être judicieusement commenté, en particulier on indiquera comment les structures de données choisies pour l'arbre et l'automate permettent de garantir les complexités trouvées.

Parsing d'un fichier de types

Le parsing d'un fichier de types pourra être fait manuellement ou en utilisant les outils de parsing du langage utilisé. Pour les outils, on propose les suivants :

— `ocamllex` et `Menhir` en OCaml
— `Lark` en Python
— `ANTLR` en Java

Cependant, la syntaxe des types est tellement simple qu'un parser écrit à la main peut être écrit sans grande difficulté. Pour gérer les parenthèses, on pourra utiliser l'algorithme dit de la gare de triage [2].

Documents XML

Pour le chargement de documents, vous pouvez soit écrire un parseur manuellement pour les documents XML simplifiés, soit utiliser une bibliothèque (xml-light en OCaml, le package `xml.dom.minidom` de la bibliothèque standard Python ou le package `javax.xml.parsers.DocumentBuilder` de la bibliothèque standard Java). On propose le code ci-dessous pour charger des documents XML.

2. https://fr.wikipedia.org/wiki/Algorithme_Shunting-yard

OCaml

En ayant installé la bibliothèque `xml-light` (support basique pour XML, un programme plus robuste devrait plutôt utiliser `ocaml-expat`, plus bas-niveau mais offrant plus de contrôle) :

```
    let doc = Xml.parse_file "fichier.xml"
    (* La variable doc contient une valeur du type Xml.xml :

    type xml =
       Element of (string * (string*string) list * xml list)
      |PCData of string

    Le premier argument de Element est l'étiquette, la liste suivante
    la liste des attributs (à ignorer) et la troisième liste la liste
    des fils.

    PCData représente des nœuds de texte, normalement non présents pour les
    fichiers du devoir.
    *)
    (* exemple: *)
    let rec hauteur d =
        match d with
          PCData _ -> 0
      | Element (_, _, children)
          1 + List.fold_left (fun acc c -> max acc (hauteur c)) 0 children
```

Python 3

En utilisant le module `dom` de la bibliothèque standard :

```
from xml.dom.minidom import parse

doc = parse("fichier.xml")
# L'objet doc contient les attributs et méthodes suivantes :
# .nodeType : un entier décrivant la sorte du type. On s'intéressera
#             uniquement aux nœuds ayant le type 1 (balises)
# .nodeName : une chaîne donnant le label du nœud
# .childNodes : un tableau des fils
### ATTENTION: pour se conformer au standard du W3C, tout document
### chargé contient un nœud fictif ayant le label '#document' qu'il
### faut ignorer. La racine se trouve dans l'attribut documentElement:

doc = doc.documentElement

def hauteur (n):
  if n.nodeType != 1:
    return 0
  h = 0
  for c in n.childNodes:
    h = max(h, hauteur(c))
  return 1  + h
```

Java

Comme Python, Java propose une implémentation du standard DOM du W3C :

```java
import org.w3c.dom.*;
import javax.xml.parsers.*;

public class test {
    public static void main(String[] args) {
      try {
          DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
          DocumentBuilder db = dbf.newDocumentBuilder();
          Node doc = db.parse("fichier.xml").getDocumentElement();
          int i = hauteur (doc);
      } catch (Exception e) {
          e.printStackTrace();
      }
    }

    static int hauteur(Node n) {
      if (n == null || n.getNodeType() != 1)
          return 0;
      int h = 0;
      NodeList nl = n.getChildNodes();
      for (int i = 0; i < nl.getLength(); i++) {
          h = Math.max(h, hauteur(nl.item(i)));
      }
      return 1 + h;
   }

}
```

6 Bonus : complexité amortie

Vous n'êtes pas obligés d'implémenter les observations faites dans cette partie pour avoir tous les points.

Si on considère la fonction `validate_opt` implémentée. On peut remarquer plusieurs choses :

1. lors de la phase top-down, étant donnée une étiquette courrante a et un ensemble d'états Q, alors l'ensemble des états Q_1 à visiter sur le sous-arbre gauche est complètement déterminé. Pltôt que de le recalculer à chaque fois, il peut être judicieux de stocker cet ensemble dans un cache (*hash-consing*). Évidemment pour celà, il faut que les fonctions de hachage et d'égalité sur la paire (a, Q) soient ls plus efficace possible, idéalement en temps constant. L'étiquette a étant une simple chaîne de caractères, il est possible de créer une fonction de hachage *parfaite* au moment du parsing du document. Pour Q, on pourra se pencher sur diverses structures de données, comme les tableaux de *bits* ou les arbres de Patricia.
2. Lors du retour de la visite du sous-arbre gauche, on a plus d'information et on peut utiliser a, Q et Q'_1, ensemble des états acceptés par le sous-arbre gauche, pour determiner Q_2
3. De la même façon, connaissant maintenant Q'_1, Q'_2 et a, Q' ensemble des états acceptés pour le nœud courant est totalement déterminé.