

# X-ENS — Info 2013

## 1. Préliminaires

**Question 1** Pour  $u = bbcbbaa$ , on a :

- (a)  $(u, 4) \models \mathbf{G}(p_a \vee p_b)$ , puisque qu'à partir de la position 4,  $u$  ne comporte que des  $a$  et des  $b$ ;
- (b)  $(u, 2) \not\models \mathbf{X}(\mathbf{G}(p_a \vee p_c))$  puisque cela signifie qu'à partir de la position 3, on ne trouve que des  $a$  et des  $c$ , ce qui est faux;
- (c)  $(u, 1) \models \mathbf{F}(\mathbf{G}(p_a \vee p_b))$ , puisque  $(u, 4) \models \mathbf{G}(p_a \vee p_b)$ ;
- (d)  $u \models (p_a \vee p_b) \mathbf{U} (p_a \vee p_c)$ , puisque  $(u, 3) \models p_a \vee p_c$  et  $(u, i) \models p_a \vee p_b$  pour  $0 \leq i < 3$ .

**Question 2** Traduit formellement, on veut :

$$\exists i < j \leq |u| - 1 : a_i = a \wedge a_j = b.$$

Cela correspond clairement à la proposition  $\mathbf{F}(p_a \wedge \mathbf{X}(\mathbf{F}p_b))$  (ou même  $\mathbf{F}(p_a \wedge \mathbf{F}p_b)$ , les lettres  $a$  et  $b$  étant distinctes). En effet, on a :

$$\begin{aligned} u \models \mathbf{F}(p_a \wedge \mathbf{X}(\mathbf{F}p_b)) &\iff \exists i \leq |u| - 1 : (u, i) \models p_a \wedge \mathbf{X}(\mathbf{F}p_b) \\ &\iff \exists i \leq |u| - 1 : a_i = a \text{ et } (u, i+1) \models \mathbf{F}p_b \\ &\iff \exists i \leq |u| - 1 : a_i = a \text{ et } \exists j \leq |u| - 1 : i < j \text{ et } a_j = b. \end{aligned}$$

**Question 3** Considérons la formule  $\text{Fin} = \neg(\mathbf{X} \mathbf{VRAI})$ . On a :

$$\begin{aligned} (u, i) \models \text{Fin} &\iff (u, i) \not\models \mathbf{X} \mathbf{VRAI} \\ &\iff (u, i+1) \not\models \mathbf{VRAI} \\ &\iff i+1 \geq |u|. \end{aligned}$$

Ayant, bien sûr,  $i \leq |u| - 1$ , on en déduit que  $(u, i) \models \text{Fin}$  si, et seulement si,  $i = |u| - 1$ .

**Question 4** À l'aide de la formule précédente, une formule qui convient clairement est :

$$\varphi = \mathbf{F}(\text{Fin} \wedge p_a).$$

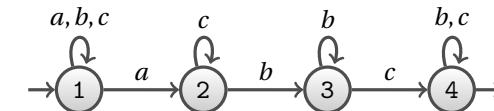
**Question 5** Définissons :

$$\begin{aligned} \varphi &= \mathbf{G}(p_a \vee p_b) \wedge \mathbf{G}(p_a \Rightarrow \mathbf{X}p_b) \wedge \mathbf{G}(p_b \Rightarrow (\text{Fin} \vee \mathbf{X}p_a)) \\ &= \mathbf{G}(p_a \vee p_b) \wedge \mathbf{G}(\neg p_a \vee \mathbf{X}p_b) \wedge \mathbf{G}(\neg p_b \vee \text{Fin} \vee \mathbf{X}p_a). \end{aligned}$$

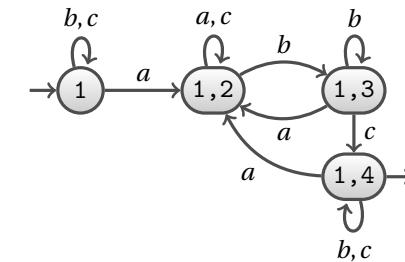
Cette formule impose que :

- le mot considéré ne comporte que des  $a$  et des  $b$ ,
  - que chaque lettre  $a$  est suivie par un  $b$  (en particulier, on ne peut finir par un  $a$ ),
  - et que chaque  $b$  qui n'est pas en fin de mot soit suivi d'un  $a$ .
- Ainsi, la formule  $\varphi$  convient.

**Question 6** Décomposons la formule proposée : la partie «  $\mathbf{F}(p_b \wedge \mathbf{X}p_c)$  » impose de trouver un  $b$  suivi d'un  $c$ . Ainsi, la partie «  $p_a \wedge \mathbf{X}(\mathbf{G}(\neg p_a)) \wedge \mathbf{F}(p_b \wedge \mathbf{X}p_c)$  » signifie que l'on trouve un  $a$  qui ne sera suivi que de  $b$  et de  $c$ , avec à un moment un  $b$  suivi d'un  $c$ . Un automate non-déterministe qui convient est donc :



**Remarque** Bien que non demandé, une version déterminisée est :



**Question 7** D'après la sémantique des formules, on a :

$$\begin{aligned} u \models \varphi \mathbf{U} \psi &\iff \exists j \leq |u| - 1 : (u, j) \models \psi \text{ et } \forall i \leq j, (u, i) \models \varphi \\ &\iff (u, 0) \models \psi \text{ ou } (\exists j \leq |u| - 1 : 1 \leq j \text{ et } (u, j) \models \psi \text{ et } \forall i \leq j, (u, i) \models \varphi) \\ &\iff u \models \psi \text{ ou } ((u, 0) \models \varphi \text{ et } \\ &\quad \exists j \leq |u| - 1 : 1 \leq j \text{ et } (u, j) \models \psi \text{ et } \forall 1 \leq i \leq j, (u, i) \models \varphi) \\ &\iff u \models \psi \text{ ou } (u \models \varphi \text{ et } u \models \mathbf{X}(\varphi \mathbf{U} \psi)) \\ &\iff u \models \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)). \end{aligned}$$

Cela prouve donc que  $\varphi \mathbf{U} \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi))$ .

## 2. Normalisation de formules

**Question 8** La fonction peut s'écrire simplement :

```
let rec taille = fun
| VRAI -> 1
| (Predicat _) -> 1
| (NON f) -> 1 + (taille f)
| (ET (f1, f2)) -> 1 + (taille f1) + (taille f2)
| (OU (f1, f2)) -> 1 + (taille f1) + (taille f2)
| (X f) -> 1 + (taille f)
| (G f) -> 1 + (taille f)
| (F f) -> 1 + (taille f)
| (U (f1, f2)) -> 1 + (taille f1) + (taille f2) ;;
```

**Question 9** Il est clair que l'on a  $F\varphi \equiv \text{VRAI} \mathbf{U} \varphi$ .

L'écriture de la fonction « `normaliseF` » s'en déduit directement :

```
let rec normaliseF = fun
| VRAI -> VRAI
| (Predicat a) -> Predicat a
| (NON f) -> NON (normaliseF f)
| (ET (f1, f2)) -> ET (normaliseF f1, normaliseF f2)
| (OU (f1, f2)) -> OU (normaliseF f1, normaliseF f2)
| (X f) -> X (normaliseF f)
| (G f) -> G (normaliseF f)
| (F f) -> U (VRAI, normaliseF f)
| (U (f1, f2)) -> U (normaliseF f1, normaliseF f2) ;;
```

Il faut en particulier faire attention, dans le cas de  $F\varphi$ , de F-normaliser  $\varphi$  en plus du traitement du connecteur F.

La complexité de l'algorithme est linéaire en la taille de la formule, puisque chaque noeud est traité exactement une fois.

**Question 10** Après la question précédente, il reste le cas de G à traiter. On peut remarquer que l'on a :

$$G\varphi \equiv \varphi \mathbf{U} (\varphi \wedge \text{Fin}) \equiv \varphi \mathbf{U} (\varphi \wedge \mathbf{X}(\neg \text{VRAI})).$$

Une autre possibilité est de remarquer (en s'inspirant de la dualité bien connue entre les connecteurs universel et existential) que  $G\varphi \equiv \neg F(\neg \varphi)$  :

$$\begin{aligned} (u, i) \models \neg F(\neg \varphi) &\iff (u, i) \not\models F(\neg \varphi) \\ &\iff \text{non}(\exists j : i \leq j \leq |u| - 1 \text{ et } (u, j) \models \neg \varphi) \\ &\iff \forall j, i \leq j \leq |u| - 1 \Rightarrow (u, j) \not\models \neg \varphi \\ &\iff \forall j, i \leq j \leq |u| - 1 \Rightarrow (u, j) \models \varphi \\ &\iff (u, i) \models G\varphi. \end{aligned}$$

On a alors :

$$G\varphi \equiv \neg(\text{VRAI} \mathbf{U} \neg \varphi).$$

La fonction « `normalise` », de complexité linéaire, s'en déduit à nouveau directement :

```
let rec normalise = fun
| VRAI -> VRAI
| (Predicat a) -> Predicat a
| (NON f) -> NON (normalise f)
| (ET (f1, f2)) -> ET (normalise f1, normalise f2)
| (OU (f1, f2)) -> OU (normalise f1, normalise f2)
| (X f) -> X (normalise f)
| (G f) -> NON (U (VRAI, NON (normalise f)))
| (F f) -> U (VRAI, normalise f)
| (U (f1, f2)) -> U (normalise f1, normalise f2) ;;
```

**Remarque** Si l'on avait utilisé une équivalence du genre  $G\varphi \equiv \varphi \mathbf{U} (\varphi \wedge \mathbf{X}(\neg \text{VRAI}))$ , pour conserver le caractère linéaire, il aurait fallu veiller à ne normaliser  $\varphi$  qu'une seule fois, même s'il apparaît deux fois dans la formule normalisé.

**Question 11** Rappelons que l'on a montré précédemment que :

$$\varphi \mathbf{U} \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)).$$

On en déduit donc la fonction « `veriteN` » :

```
let rec veriteN u i = fun
| VRAI -> i < (string_length u)
| (Predicat a) -> u.[i] = a
| (NON f) -> not (veriteN u i f)
| (ET (f1, f2)) -> (veriteN u i f1) && (veriteN u i f2)
| (OU (f1, f2)) -> (veriteN u i f1) or (veriteN u i f2)
| (X f) -> veriteN u (i + 1) f
| (G f) -> failwith "Formule non normalisée"
| (F f) -> failwith "Formule non normalisée"
| (U (f1, f2)) ->
  (veriteN u i f2) or (
    (veriteN u i f1) &&
    (veriteN u (i + 1) (U (f1, f2)))
  ) ;;
```

La terminaison de cette fonction est assurée, puisque chaque appel récursif se fait avec une valeur pour  $i$  strictement supérieure, ou avec une formule de taille strictement inférieure.

Concernant la complexité, lors de l'appel « `veriteN u i f` », on peut remarquer que les appels récursifs à la fonction « `veriteN` » se font au plus une fois pour chaque sous-formule et chaque position  $j \geq i$ . On en déduit, chaque appel à cette fonction étant traité en temps constant, que la complexité est en  $O(|\varphi| \times (|u| - i + 1))$ .

### 3. Rationalité des langages décrits par des formules

**Question 12** La fonction « `initialise` » ne pose pas de problème, tous les booléens devant être initialisée à « `false` ».

```
let rec initialise = fun
| VRAI -> (AVRAI, false)
| (Predicat a) -> ((APredicat a), false)
| (NON f) -> (ANON (initialise f), false)
| (ET (f1, f2)) -> (AET (initialise f1, initialise f2), false)
| (OU (f1, f2)) -> (AOU (initialise f1, initialise f2), false)
| (X f) -> (AX (initialise f), false)
| (G _) -> failwith "Formule non normalisée"
| (F _) -> failwith "Formule non normalisée"
| (U (f1, f2)) -> (AU (initialise f1, initialise f2), false) ;;
```

**Question 13** On a les équivalences suivantes :

$$\begin{aligned}
 \mathbf{VRAI} &\in S' \\
 p_b \in S' &\iff a = b \\
 \neg\varphi \in S' &\iff \varphi \notin S' \\
 \varphi \wedge \psi \in S' &\iff (\varphi \in S' \text{ et } \psi \in S') \\
 \varphi \vee \psi \in S' &\iff (\varphi \in S' \text{ ou } \psi \in S') \\
 \mathbf{X}\varphi \in S' &\iff \varphi \in S \\
 \varphi \mathbf{U} \psi \in S' &\iff (\psi \in S' \text{ ou } (\varphi \in S' \text{ et } \varphi \mathbf{U} \psi \in S))
 \end{aligned}$$

Elles recouvrent tous les cas possibles, et montrent que l'appartenance d'une formule  $\varphi$  à  $S'$  dépendant uniquement de l'appartenance de sous-formules strictes de  $\varphi$  à  $S'$ , ou de l'appartenance de formules à  $S$ . Autrement dit, la détermination de  $S'$  ne dépend bien que de  $a$  et de  $S$ .

**Question 14** Les équivalences précédentes donnent directement la fonction suivante :

```
let rec maj phi a =
  match phi with
  | (AVRAI, _) -> (AVRAI, true)
```

```

| ((APredicat b), _) -> ((APredicat b), a = b)
| (ANON f, _) ->
  let (phi', b') = maj f a in
  (ANON (phi', b'), not b')
| ((AET (f1, f2)), _) ->
  let (phi1', b1) = maj f1 a
  and (phi2', b2) = maj f2 a in
  (AET ((phi1', b1), (phi2', b2)), b1 && b2)
| ((AOU (f1, f2)), _) ->
  let (phi1', b1) = maj f1 a
  and (phi2', b2) = maj f2 a in
  (AOU ((phi1', b1), (phi2', b2)), b1 or b2)
| (AX (phi, b), _) ->
  let (phi', b') = maj (phi, b) a in
  (AX (phi', b'), b)
| (AU (f1, f2), b) ->
  let (phi1', b1) = maj f1 a
  and (phi2', b2) = maj f2 a in
  (AU ((phi1', b1), (phi2', b2)), b2 or (b1 && b)) ;;

```

La remarque de la question précédente permet d'affirmer que la fonction termine. De plus, sa complexité est linéaire en la taille de la formule, le booléen attaché à chaque nœud se calculant en temps constant.

**Question 15** La fonction « `sousFormulesVraies` » s'obtient en itérant la fonction précédente sur les différentes lettres du mot, de droite à gauche.

```
let sousFormulesVraies f s =
  let e = ref (initialise f)
  and p = ref (string_length s) in
  while (!p > 0) do
    decr p ;
    e := maj !e s.[!p]
  done ;
  !e ;;
```

Le corps de la boucle est linéaire en la taille  $|\varphi|$  de la formule  $\varphi$ , et la fonction répète  $|u|$  fois la boucle. La complexité de la fonction est donc en  $O(|u| \times |\varphi|)$ .

**Question 16** On en déduit directement :

```
let veriteBis f s =
```

```
let (_, b) = sousFormulesVraies f s in
b ;;
```

De façon plus compacte, on peut même écrire :

```
let veriteBis f s =
  snd (sousFormulesVraies f s) ;;
```

**Question 17** D'après les questions précédentes, on construit simplement un automate complet déterministe reconnaissant  $\tilde{L}_\varphi$  pour une formule  $\varphi$  donnée. En effet, si  $F(\varphi)$  est l'ensemble des sous-formules de  $\varphi$ , on définit l'automate déterministe  $\mathcal{A} = (Q, \delta, i_0, F)$  avec :

- $Q = \wp(F(\varphi))$  : les états de  $\mathcal{A}$  sont des ensembles de sous-formules de  $\varphi$ ;
- la fonction de transition  $\delta$  est exactement la fonction « `maj` » définie précédemment;
- $i_0 = \emptyset$  : avant d'avoir lu un caractère, aucune sous-formule n'est vérifiée;
- $F = \{S \in Q \mid \varphi \in S\}$  : les états finaux sont, en s'inspirant de « `veriteBis` », ceux pour lesquels  $\varphi$  est vérifiée.

Cet automate a  $\text{Card } \wp(F(\varphi)) = 2^{|\varphi|}$  états, avec  $|\varphi|$  désignant la taille de  $\varphi$  (ou moins d'états si on l'émondre).

**Question 18** Pour obtenir un automate reconnaissant  $L_\varphi$ , il suffit de considérer l'automate précédent (qui reconnaît  $\tilde{L}_\varphi$ ) et de "retourner" toutes les flèches (et, bien sûr, d'intervertir les états initiaux et finaux). Cet automate, a priori non déterministe, a  $2^{|\varphi|}$  états.

## 4. Satisfiabilité et expressivité

**Question 19** Pour obtenir l'ensemble des états atteignables d'un automate déterministe, on effectue un parcours du graphe sous-jacent. Par exemple, un parcours en profondeur s'effectue ainsi :

1. Marquer tous les sommets comme « `non vu` »;
2. Définir la procédure « `traiter s` » où « `s` » désigne un sommet de la façon suivante :
  - (a) Marquer « `s` » comme `vu`;
  - (b) Appeler récursivement « `traiter t` » pour tout successeur « `t` » de « `s` » qui est marqué comme « `non vu` ».
3. Appeler « `traiter i0` ».
4. Retourner la liste des sommets marqués comme « `vu` ».

**Question 20** Considérons un automate  $\mathcal{A}$  reconnaissant  $L_\varphi$  (comme celui obtenu à la question 18). Un mot  $u_{\min}$  de longueur minimale reconnu par  $\mathcal{A}$  est l'étiquette d'un

chemin passant au plus une fois par chaque sommet. Sa longueur est donc majorée par  $2^{|\varphi|} - 1$ .

**Question 21** La clé ici est d'avoir majoré la taille maximale d'un mot  $u$  de longueur minimale satisfaisant une formule  $\varphi$ . Cela rend le problème de la satisfiabilité de  $\varphi$  décidable.

Pour procéder, les questions précédentes suggèrent de "construire" l'automate reconnaissant  $\tilde{L}_\varphi$  et de chercher un état final atteignable. Bien sûr, on ne construira pas l'automate à l'avance, mais au fur et à mesure, en même temps que la recherche d'un état final.

On peut procéder ainsi, effectuant un parcours du graphe :

1. on considère deux ensembles : « `vu` », initialement vide, et « `bordure` » qui ne contient au début que la paire état initial - mot vide  $(i_0, \varepsilon)$ ;
2. répéter tant que « `bordure` » est non vide :
  - (a) soit  $(s, u)$  un élément que l'on retire de « `bordure` »;
  - (b) si  $\varphi \in s$ , on termine en renvoyant le mot  $u$ ;
  - (c) sinon :
    - i. on ajoute  $s$  à « `vu` »;
    - ii. si  $s_a = \text{maj}(s, 'a')$  n'appartient pas à « `vu` », on l'ajoute à « `bordure` »;
    - iii. on fait de même pour  $s_b = \text{maj}(s, 'b')$
3. si « `bordure` » devient vide sans que l'algorithme ait terminé plus tôt, alors la formule n'est pas satisfiable.

Pour un parcours en profondeur, l'ensemble « `bordure` » est représenté par une liste. Les ajouts et retraits en tête se font en temps constant. Supposons dans une première analyse que « `vu` » est aussi représenté par une liste.

La boucle principale va être effectuée au plus  $O(2^{|\varphi|})$  fois. À chaque fois, on calcule un successeurs ( $s_a$  puis  $s_b$ ), en  $O(|\varphi|)$ , puis on teste son appartenance éventuelle à « `vu` ». La longueur de « `vu` » est au plus  $O(2^{|\varphi|})$ , chaque test d'égalité se faisant en  $O(|\varphi|)$ , le test d'appartenance se fait donc en  $O(|\varphi|2^{|\varphi|})$  (l'appel de « `maj` » étant négligeable devant ce dernier).

La satisfiabilité de  $|\varphi|$  se décide donc en  $O(|\varphi|2^{2|\varphi|})$ . D'après l'énoncé, on pose donc  $\alpha = 2$  et  $\beta = 1$ , l'absence du terme  $|\varphi|$  étant probablement une petite erreur de l'énoncé (mais la formule est correcte pour tout  $\beta > 1$ ).

On peut sensiblement améliorer les choses en utilisant pour « `vu` » un arbre binaire équilibré. Dans ce cas, le test d'appartenance comme l'insertion se font maintenant en  $O(\log_2 2^{|\varphi|}) = O(|\varphi|)$  comparaisons. Une itération se fait maintenant en  $O(|\varphi|^2)$  et la satisfiabilité d'une formule  $\varphi$  en  $O(|\varphi|^2 2^{|\varphi|})$ .

En s'éloignant du programme, on peut considérer l'usage d'une table de hachage où les opérations se font en temps constant en moyenne. On obtient alors une complexité de  $O(|\varphi|2^{|\varphi|})$ .

```

let satisfiable f =
  let rec boucle vu bordure =
    match bordure with
    [] -> "Formule non satisfiable"
  | (s, u) :: bordure' ->
    let (_, b) = s in
    if b then u else
    let vu' = s :: vu in
    let sa = maj s 'a' in
    let bordure2 =
      if (mem sa vu')
      then bordure'
      else (sa, "a" ^ u) :: bordure'
    in
    let sb = maj s 'b' in
    let bordure3 =
      if (mem sb vu')
      then bordure2
      else (sb, "b" ^ u) :: bordure2
    in
    boucle vu' bordure3
  in
  boucle [] [initialise f, ""]
;;

```

FIGURE 1 – Fonction « satisfiable »

Prouvons cela par induction structurelle. Tout d'abord, on a clairement

$$\begin{array}{ll}
 E_{\text{VRAI}} = \mathbb{N} & E_{p_a} = \mathbb{N} \\
 E_{\varphi \wedge \psi} = E_{\varphi} \cap E_{\psi} & E_{\varphi \vee \psi} = E_{\varphi} \cup E_{\psi} \\
 E_{\neg \varphi} = \mathbb{N} \setminus E_{\varphi} & E_{\mathbf{X}\varphi} = \{i + 1 \mid i \in E_{\varphi}\}
 \end{array}$$

Dans tous ces cas, notre hypothèse est bien vérifiée. Reste à traiter le cas  $\varphi \mathbf{U} \psi$ . Puisque l'on a

$$\varphi \mathbf{U} \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)),$$

on en déduit que

$$i \in E_{\varphi \mathbf{U} \psi} \iff [i \in E_{\psi} \text{ ou } (i \in E_{\varphi} \text{ et } i - 1 \in E_{\varphi \mathbf{U} \psi})].$$

Ainsi, pour tout  $i \in \mathbb{N}^*$ , on a

$$i \in E_{\varphi \mathbf{U} \psi} \iff \exists j \leq i : j \in E_{\psi} \text{ et } \{j + 1, j + 2, \dots, i - 1, i\} \subseteq E_{\varphi}.$$

Cela assure que  $E_{\varphi \mathbf{U} \psi}$  est aussi de la bonne forme, ce qui achève de prouver notre résultat.

Ainsi, il n'existe pas de formule  $\varphi$  telle que  $E_{\varphi} = 2\mathbb{N}^*$

La programmation proprement dite de cette fonction (avec la version liste) est donnée figure 1.

**Question 22** Nous allons montrer tout d'abord que, pour tout formule  $\varphi$ , l'ensemble

$$E_{\varphi} = \{i \in \mathbb{N} \mid a^i \models \varphi\}$$

est soit une partie finie de  $\mathbb{N}$ , soit la réunion d'une partie finie de  $\mathbb{N}$  et d'un intervalle d'entiers de la forme  $\{k \in \mathbb{N} \mid k \geq n\}$ .