Projet LO21: Rapport final

Adrien Burgun

Automne 2020

Résumé

Le projet de ce semestre pour le cours de **LO21** (Algorithmique et Programmation II) porte sur un *« système expert »*. Un système expert est constitué de 3 éléments :

Une base de connaissance, qui prend la forme suivante :

$$A \wedge B \wedge ... \wedge Z \Rightarrow \Omega$$

Où $A, B, ...$ sont les symboles (d'arité zéro, aussi appelés « propositions ») constituant la *prémisse* et Ω est la *conclusion.*

Une base de faits, qui est la liste des symboles ayant la valeur *« Vrai »* (qui correspond à l'état *« Certain »*).
Un symbole ne faisant pas partie de cette liste a par défaut la valeur *« Faux »* (qui correspond à l'état *« Incertain »*).

Un moteur d'inférence, qui, à partir de la base de connaissance et la base de faits, déduit quels autres symboles sont aussi vrais et les ajoute à la base de faits.

Nous définirons d'abords le type *« Règle »*, constituant la base de connaissance. Nous définirons ensuite le type *« BC »* (Base de Connaissance).
Nous décrirons enfin le moteur d'inférence comme implémenté dans ce projet, avec différents exemples.

1 Règles

Soit **Règle** le type représentant une règle sous la forme d'une liste chaînée de symboles :

Structure 1 : Règle		
Nom	**Type**	**Description**
symbole	Règle → Chaîne de charactères	Retourne le nom du symbole correspondant au noeud en tête de liste.
suivant	Règle → Règle	Retourne une référence au prochain élément de la liste chaînée, *règle_vide* si l'élément est le dernier de la liste.
nouvelle_règle	((Chaîne de charactères) × Règle) → Règle	Compose une nouvelle règle à partir du nom d'un symbole et une référence à la prochaine règle.
règle_vide	Règle	La règle vide.
mettre_suivant	(Règle × Règle) → Règle	Modifie une Règle pour y attacher une Règle comme règle suivante ; retourne également la règle modifiée.

Le dernier élément d'une telle liste chaînée correspond à la conclusion de la règle, tandis que tous les autres éléments appartiennent à la prémisse (contrainte du projet).

Les axiomes sur ces fonctions sont :

- *symbole*(*nouvelle_règle*(s, r)) = s
- *suivant*(*nouvelle_règle*(s, r)) = r
- A ⟵ *nouvelle_règle*(s, r) ; *mettre_suivant*(a, r') ⇒ *suivant*(A) = r'

1.1 Créer une règle vide

Nous représentons une règle vide par *règle_vide*. Voici l'algorithme permettant de créer une règle vide :

Algorithme 1 : RègleVide

```
   Variables : R: La règle vide à retourner
   Résultat : R: Règle
1  Début RègleVide()
2  |  R ⟵ règle_vide
3  Fin
```

1.2 Ajouter une proposition à la prémisse d'une règle

L'ajout des propositions (symboles) à la prémisse d'une règle se fait par l'algorithme *AjoutPrémisse* défini ci-dessous. Cet ajout se fait en queue de la liste chaînée (contrainte du projet). La liste chaînée donnée en entrée est modifiée par l'algorithme et est retournée.

Algorithme 2 : AjoutPrémisse

Variables : R: La règle à modifier
R': Une variable temporaire pour traverser la liste chaînée
symbole: Le nom de la proposition (symbole) à insérer
Données : R: Règle, *symbole*: Chaîne de charactères
Résultat : R: Règle

```
1  Début AjoutPrémisse(R, symbole)
2      Si R = règle_vide alors
3          R ⟵ nouvelle_règle(symbole, règle_vide)
4      Sinon
5          R' ⟵ R
           // Répeter jusqu'à ce qu'on atteigne le dernier élément
6          Tant que suivant(R') ≠ règle_vide faire
7              R' ⟵ suivant(R')
8          Fin
           // R' contient désormais le dernier élément de la liste
9          mettre_suivant(R', nouvelle_règle(symbole, règle_vide))
10     FinSi
11 Fin
```

1.3 Créer la conclusion d'une règle

Créer la conclusion d'une règle revient à ajouter une proposition (symbole) à la fin de la règle. Pour ce faire, nous ré-utilisons l'algorithme AjoutPrémisse défini plus tôt.

Algorithme 3 : AjoutConclusion

Variables : R: La règle à modifier
symbole: Le nom de la proposition (symbole) à insérer comme conclusion
Données : R: Règle, *symbole*: Chaîne de charactères
Résultat : R: Règle

```
1  Début AjoutConclusion(R, symbole)
2      R ⟵ AjoutPrémisse(R, symbole)
3  Fin
```

1.4 Tester si une proposition appartient à la prémisse d'une règle

Nous testons si une proposition appartient à la prémisse d'une règle en traversant celle-ci de manière récursive (contrainte du projet).

Les 3 cas minimaux sont :

$R =$ **[]** (règle vide) : retourner « Faux »
$R =$ **{symbole : "...", suivant : *règle_ vide*}** (conclusion) : retourner « Faux »
$R =$ **{symbole : symbole_recherché, suivant : ...}** (symbole trouvé) : retourner « Vrai »

Dans les autres cas, nous retournons de manière récursive le résultat de la même fonction, appelée sur *suivant*(R).

Algorithme 4 : TestAppartenance

Variables : R: La règle à étudier
symbole: La nom de la proposition à rechercher
résultat: Si oui ou non la proposition à été trouvée
Données : R: Règle, *symbole*: Chaîne de charactères
Résultat : *résultat*: Booléen

```
1  Début TestAppartenance(R, symbole)
2  |  Si R = règle_vide alors
   |  |  // Règle vide
3  |  |  résultat ⟵ Faux
4  |  Sinon si suivant(R) = règle_vide alors
   |  |  // Conclusion
5  |  |  résultat ⟵ Faux
6  |  Sinon si symbole(R) = symbole alors
   |  |  // Symbole trouvé
7  |  |  résultat ⟵ Vrai
8  |  Sinon
9  |  |  résultat ⟵ TestAppartenance(suivant(R), symbole)
10 |  FinSi
11 Fin
```

1.5 Supprimer une proposition de la prémisse d'une règle

Nous supprimons une proposition de la prémisse d'une règle de manière récursive. Les cas minimaux sont les suivants :

$R =$ **[]** (règle vide) : retourner *règle_ vide*
$R =$ **{symbole : "...", suivant : *règle_ vide*}** (conclusion) : retourner R

Dans le cas général, nous attribuons à *suivant*(R) la valeur retournée par cette fonction, appelée sur *suivant*(R), et nous retournons *suivant*(R) si le noeud correspond au

symbole et R sinon.

Algorithme 5 : SupprimerSymbole

Variables : R: La règle à modifier
symbole: La nom de la proposition à rechercher
R': La règle privée de *symbole* dans sa prémisse
Données : R: Règle, *symbole*: Chaîne de charactères
Résultat : R': Règle

```
1  Début SupprimerSymbole(R, symbole)
2      Si R = règle_vide alors
           // Règle vide
3          R' ⟵ règle_vide
4      Sinon si suivant(R) = règle_vide alors
           // Conclusion
5          R' ⟵ R
6      Sinon
7          Si symbole(R) = symbole alors
               // Retourner le reste de la liste, sans ce noeud
8              R' ⟵ SupprimerSymbole(suivant(R), symbole)
9          Sinon
10             R' ⟵ mettre_suivant(R, SupprimerSymbole(suivant(R), symbole))
11         FinSi
12     FinSi
13 Fin
```

1.6 Tester si la prémisse d'une règle est vide

Voici la fonction retournant « Vrai » si la prémisse d'une règle est vide et « Faux » si la prémisse d'une règle contient au moins 1 symbole :

Algorithme 6 : PrémisseVide

Variables : R: La règle à étudier
résultat: Si oui ou non la prémisse d'une règle est vide
Données : R: Règle
Résultat : *résultat*: Booléen

```
1 Début PrémisseVide(R)
2     Si R = règle_vide alors
          // La règle est vide, donc sa prémisse est vide
3         résultat ⟵ Vrai
4     Sinon si suivant(R) = règle_vide alors
          // La règle n'a qu'une conclusion, donc sa prémisse est vide
5         résultat ⟵ Vrai
6     Sinon
7         résultat ⟵ Faux
8     FinSi
9 Fin
```

1.7 Accéder à la proposition se trouvant en tête d'une prémisse

Voici la fonction retournant la valeur de la proposition se trouvant en tête d'une prémisse. Si la prémisse est vide, alors la fonction retourne *règle_vide*.

Algorithme 7 : PremierSymbole

Variables : R: La règle à étudier
résultat: La valeur du premier symbole de la prémisse, si existant
Données : R: Règle
Résultat : *résultat*: Chaîne de charactères

```
1 Début PremierSymbole(R)
2     Si PrémisseVide(R) alors
          // La prémisse est vide: nous retournons règle_vide
3         résultat ⟵ règle_vide
4     Sinon
5         résultat ⟵ symbole(R)
6     FinSi
7 Fin
```

1.8 Accéder à la conclusion d'une règle

La conclusion se trouvant à la fin d'une règle, nous traversons simplement la liste chaînée jusqu'au dernier élément de celle-ci. Si la liste est vide, alors la fonction retourne *règle_vide*.

Algorithme 8 : ConclusionRegle

Variables : R: La règle à étudier
R': Variable temporaire pour traverses la liste chaînée
résultat: La valeur du premier symbole de la prémisse, si existant
Données : R: Règle
Résultat : *résultat*: Chaîne de charactères

```
1  Début ConclusionRegle(R)
2      Si suivant(R) = règle_vide alors
3          résultat ⟵ règle_vide
4      Sinon
5          R' ⟵ R
           // Avançer jusqu'à la fin de la liste
6          Tant que suivant(R') ≠ règle_vide faire
7              R' ⟵ suivant(R')
8          Fin
9          résultat ⟵ symbole(R')
10     FinSi
11 Fin
```

2 Base de Connaissance

Soit **BC** le type représentant une base de connaissance ; celle-ci prend la forme d'une liste chaînée de Règles :

Structure 2 : BC

Nom	**Type**	**Description**
règle	BC → Règle	Retourne une référence à la règle correspondant à ce noeud.
suivant	BC → BC	Une référence au prochain élément de la liste chaînée, NULL si l'élément est le dernier de la liste.
nouvelle_base	(Règle BC) → BC	Crée une nouvelle base à partir d'une référence vers une Règle et d'une référence vers BC
base_vide	BC	La base vide

Les axiomes sur ces fonctions sont :

— $règle(nouvelle\_base(\mathrm{r}, \mathrm{b})) = \mathrm{r}$
— $suivant(nouvelle\_base(\mathrm{r}, \mathrm{b})) = \mathrm{b}$

2.1 Créer une base vide

Nous représentons une base vide par *base_ vide*. Voici l'algorithme permettant de créer une base vide :

Algorithme 9 : BaseVide

Variables : B: La base vide à retourner
Résultat : B: BC
1 **Début** BaseVide()
2 | $B \longleftarrow base\_vide$
3 **Fin**

2.2 Ajouter une règle à une base de connaissance

L'ajout de règle à la base de connaissance se fait en tête. Voici son algorithme :

Algorithme 10 : AjoutRègle

Variables : B: La base de connaissance à modifier
règle: La valeur de *règle* à mettre dans le noeud
B': La base de connaissance contenant la nouvelle règle
Données : B: BC ; *règle*: Règle
Résultat : B': BC
1 **Début** AjoutRègle(B, *règle*)
2 | $B' \longleftarrow nouvelle\_base(règle, B)$
3 **Fin**

2.3 Accéder à la règle se trouvant en tête de la base

L'ajout de règle à la base de connaissance se fait en tête. Voici son algorithme :

Algorithme 11 : TêteBase

Variables : B: La base de connaissance à modifier
R: La règle se trouvant en tête de la base, *règle_vide* si la base est vide
Données : B: BC
Résultat : R: Règle

```
1 Début TêteBase(B)
2 |  Si suivant(B) = base_vide alors
3 |  |  R ⟵ règle_vide
4 |  Sinon
5 |  |  R ⟵ règle(B)
6 |  FinSi
7 Fin
```