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 de symboles :

Structure 1 : Règle		
Nom	**Type**	**Description**
symbole	Règle → Symbole	Retourne le nom du symbole correspondant au noeud en tête de liste.
suivant	Règle → Règle	Retourne une référence au prochain élément de la liste, *règle_vide* si l'élément est le dernier de la liste.
nouvelle_règle	((Symbole) × 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 type « Symbole » correspond dans l'implémentation C à une chaîne de caractères. Le dernier élément d'une telle liste 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'

Nous pouvons noter que :

Liste ≺ Règle [∅ : *règle_vide*, tête : *symbole*, reste : *suivant*, insérer_tête : *nouvelle_règle*, Élement : Symbole]

(Un type concret implémentant Règle implémente également Liste)

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
Début RègleVide()
  $R \longleftarrow$ *règle_vide*
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 (contrainte du projet). La liste 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
- *symbole*: Le nom de la proposition (symbole) à insérer

Données : R: Règle, *symbole*: Symbole
Résultat : R: Règle
Assertion : R n'a pas encore de conclusion

```
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*: Symbole
Résultat : R: Règle
Assertion : R n'a pas encore de conclusion

```
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*: Symbole
Résultat : *résultat*: Booléen
Assertion : R a une conclusion

```
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*: Symbole
Résultat : R': Règle
Assertion : R a une conclusion

```
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
Assertion : R a une conclusion ou R est vide

```
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 : TêteRègle

Variables :
- R: La règle à étudier
- *résultat*: La valeur du premier symbole de la prémisse, si existant

Données : R: Règle
Résultat : *résultat*: Symbole
Assertion : R a une conclusion

```
1 Début TêteRègle(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 jusqu'au dernier élément de celle-ci. Si la liste est vide, alors la fonction retourne *règle_vide*.

Algorithme 8 : ConclusionRègle

Variables :

- R: La règle à étudier
- R': Variable temporaire pour traverses la liste
- *résultat*: La valeur du premier symbole de la prémisse, si existant

Données : R: Règle
Résultat : *résultat*: Symbole
Assertion : R a une conclusion

```
1  Début ConclusionRègle(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 (liste de règle) ; celle-ci prend la forme d'une liste 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, 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*(r, b)) = r
— *suivant*(*nouvelle_ base*(r, b)) = 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' ⟵ nouvelle_base(règle, B)
3 Fin
```

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

Voici l'algorithme permettant d'accéder à la règle se trouvant en tête :

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 B = base_vide alors
3 |  |  R ⟵ règle_vide
4 |  Sinon
5 |  |  R ⟵ règle(B)
6 |  FinSi
7 Fin
```

3 Moteur d'inférence

Le moteur d'inférence est un algorithme permettant de déduire à partir de la liste initiale des symboles (propositions) ayant la valeur « Vrai » et de la base de connaissance la liste de tous les symboles étant vrais.

L'algorithme éxecute un maximum de n (la longueur de la base de connaissance) passages sur les règles, celles-cis prenant la forme suivante ($p \geq 0$ symboles S_x dans la prémisse et le symbole Ω_k dans la conclusion) :

$$BC \vdash (\bigwedge_{x=0}^{x<p} S_x) \Rightarrow \Omega_k$$

$$\text{Avec :} \quad (\bigwedge_{x=0}^{x<p} S_x) = \begin{cases} S_0 \wedge S_1 \wedge ... \wedge S_{p-1} & \text{Si } p > 0 \\ Vrai & \text{Si } p = 0 \end{cases}$$

Si Ω_k n'appartient pas encore à la liste des symboles vrais et que $\forall x \in [0, p[, S_x = Vrai$ ou $p = 0$, alors on ajoute Ω_k à la liste des symboles vrais.

Algorithme 12 : MoteurInférence

Variables :

- B: La base de connaissance
- B': Variable temporaire pour traverser la base de connaissance
- S: La liste des symboles initialements vrais
- S': La liste des symboles initialement vrais ainsi que ceux déduits des règles d'induction
- *ajouté*: Si oui ou non un symbole à été rajouté à S' dans le dernier passage

Données : B: BC ; S: Liste(Symbole)
Résultat : S': Liste(Symbole)
Assertion : $\forall R \in B, R \neq règle\_vide$

```
1  Début MoteurInférence(B, S)
2      S' ⟵ S
       // Boucle principale: Répétée jusqu'à ce qu'il n'y aie plus de
          modification à S' possibles
3      Faire
4          ajouté ⟵ Faux
5          B' ⟵ B
           // Pour toutes les règles de la base de connaissance...
6          Tant que B' ≠ base_vide faire
               // Si la prémisse n'est pas vide et la conclusion
                  n'appartient pas à S'...
7              Si non(ListeContient(S', ConclusionRègle(TêteBase(B')))) alors
                   // Si la prémisse est vraie, alors on ajoute la
                      conclusion à S'
8                  Si PrémisseVraie(TêteBase(B'), S') alors
9                      S' ⟵ insérer_tête(S', ConclusionRègle(TêteBase(B')))
10                     ajouté ⟵ Vrai
11                 FinSi
12             FinSi
13             B' ⟵ suivant(B')
14         Fin
15     Tant que ajouté
16 Fin
```

Algorithme 13 : ListeContient

Variables :

- T: Un type d'objets comparables
- L: Une liste d'objets de type T
- E: La valeur à trouver dans L
- *résultat*: Si oui ou non E est trouvé dans L

Données : L: Liste(T) ; E: T
Résultat : *résultat*: Booléen

```
1  Début ListeContient(L, E)
2      Si est_vide(L) alors
3          résultat ⟵ Faux
4      Sinon si tête(L) = E alors
5          résultat ⟵ Vrai
6      Sinon
7          résultat ⟵ ListeContient(reste(L), E)
8      FinSi
9  Fin
```

Algorithme 14 : PrémisseVraie

Variables :

- S: La liste de symboles vrais
- R: La règle à vérifier
- R': Variable utilisée pour traverser R
- *résultat*: Si oui ou non la prémisse de la règle est vraie

Données : R: Règle ; S: Liste(Symbole)
Résultat : *résultat*: Booléen
Assertion : $R \neq$ *règle_vide*

```
1  Début PrémisseVraie(R, S)
2      résultat ⟵ Vrai
3      Si non(PrémisseVide(R)) alors
4          R' ⟵ R
5          Tant que suivant(R') ≠ règle_vide faire
6              Si non(ListeContient(S, TêteRègle(R'))) alors
7                  résultat ⟵ Faux
8              FinSi
9              R' ⟵ suivant(R')
10         Fin
11     FinSi
12 Fin
```