Université de Montpellier
L2 informatique

Démonstrateur automatique propositionnel

Rapport de Projet T.E.R
Projet informatique - HLIN405
Année 2019-2020

Étudiants
Jérémy Laurès
Cibo Chen
Fei Yang
Merwane Rakkaoui

Encadrant
Christian Retoré

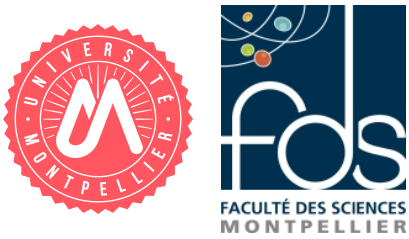


Table des matières

Chapitre 1

Introduction

S'il est venu Seul, il a pris le Bus ou le Train. S'il a pris le Bus ou son Automobile alors il est arrivé en Retard et il a Manqué la réunion. Il n'est pas arrivé en Retard.

Peut-on en conclure que « s'il est venu Seul, il a pris le Train » ?

Un problème de logique tel que celui ci-dessus est résolvable avec un démonstrateur automatique propositionnel. C'est un programme qui lorsqu'on lui donne une formule recherche une preuve et renvoie une preuve si la formule est vraie, si ce n'est pas démontrable le programme renvoie aussi les valeurs de vérité qui rendent la formule fausse.

Il existe un algorithme assez simple pour déterminer si une proposition complexe est ou non conséquence d'une autre, ou qu'une formule est une tautologie, comme l'a montré Stephen C. Kleene dans les années cinquante. Cet algorithme est basé sur un système de preuve particulier, le calcul des séquents réversible. L'algorithme « remonte » les règles de ce calcul en décomposant les formules. Lorsque l'algorithme termine, on dispose soit d'une preuve formelle dans le calcul des séquents réversible, soit d'une assignation de valeurs de vérités VRAI et FAUX aux propositions atomiques qui rendent FAUX ce que nous cherchions à établir.

Un tel problème de décision étant NP complet en fonction de la taille de la formule à prouver ou à réfuter, cet algorithme ne sera pas plus rapide dans le pire des cas que l'écriture d'une table de vérité à 2p lignes où p est le nombre de propositions atomiques. Cependant, sur l'exemple ci-dessus, la méthode des tables de vérité utilise une table à 896 cases (64 lignes par 14 colonnes) alors que l'algorithme ici proposé répond en 6 étapes, en produisant une preuve formelle comportant 6 règles du calcul des séquents.

Le projet consiste à programmer cet algorithme de recherche de preuve et de recherche de réfutation, en réfléchissant bien aux structures de données et au langage de programmation utilisé (qui n'est pas imposé) pour représenter les séquents.

Le problème étudié ici est très adapté à notre parcours d'étude puisqu'il correspond à une partie du programme d'un module de licence 3 informatique (Logique 2), et à des ressemblance au module précédent qui est en licence 2 informatique (Logique 1). Dans le monde

de l'informatique la recherche de preuve est très utilisée notamment dans l'intelligence artificielle, qui elle même est présente dans de nombreux domaines et dans le monde qui nous entoure, et surtout en vérification de programme pour le génie logiciel.

Lors de ce projet, notre objectif est de créer un programme de recherche de preuve qui une fois la formule de base rentrée nous renverra la preuve ou les valeurs de vérité produisant un contre-exemple. Le but de ce projet étant non seulement de fournir à l'utilisateur la réponse au problème posé mais aussi une explication du résultat trouvé par la machine.

Si le temps nous le permet nous avons comme objectif de faire une interface d'entrée et de sortie, pour rendre la preuve lisible.

Notre cahier des charges est simple, il nous faut un programme prenant un séquent en entrée qui calcule la preuve à l'aide de règles et renvoie la preuve ou, si il n'y a pas de preuve, les valeurs de vérité des propositions d'un contre exemple.

Pour cela nous décidons d'utiliser le langage de programmation Racket qui nous permettra d'utiliser les listes plus facilement.

Une bonne répartition du travail et l'implication de tout le monde nous a permis de mener à bien ce projet.

Chapitre 2

Organisation du projet

2.1 Organisation du travail

Lors de ce projet, comme nous sommes un groupe de quatre et que nous sommes deux en licence Informatique (et dans le même groupe) ainsi que deux en licence Mathématiques-Informatique (dans le même groupe également), nous nous voyions (avant le 16 mars) tous les jours de la semaine. Nous nous tenions informés de l'avancée de chacun pour pouvoir mieux avancer en groupe.

Pendant le confinement il était logiquement impossible de se voir et donc nous nous sommes réunis tout les deux ou trois jours pour faire un point rapide et continuer d'avancer ensemble. Si un problème se présentait, nous pouvions ainsi le régler rapidement.

Toutes les deux semaines puis chaque semaine pendant le confinement, nous nous sommes fixés un rendez-vous avec notre encadrant, Christian Retoré, afin de faire le point et de pouvoir bénéficier des son aide pour mieux comprendre et faire la recherche de preuve.

2.2 Répartition du travail

Pour ce projet nous avons du répartir les taches, mais d'abord il a fallu que tout le monde comprenne comment fonctionne le calcul des séquents pour la logique classique propositionnelle. Nous avons donc été obligés de comprendre le fonctionnement et d'apprendre les règles avant de pouvoir commencer. Par la suite, une fois cette grosse étape passée, le travail de codage pouvait commencer.

Le plus important et le plus dur étant le code de recherche de preuve dans un langage que nous avions jamais utilisé jusque là, nous avons décidé d'y travailler à quatre dessus. Au bout d'un moment il a fallu commencer le rapport, nous avons donc séparé le groupe, une personne s'occupait de commencer le rapport tandis que les trois autres continuaient de faire le programme. Arrivé au stade où le programme était bientôt fini, une deuxième personne est passée sur le rapport pour avancer plus rapidement et efficacement, pour au final y être à quatre une fois le programme fini.

2.3 Outils utilisés

Pour faire ce projet nous avons utilisé,

DrRacket, un environnement de programmation permettant de coder avec le langage de programmation Racket [**?**].

Racket est un langage de programmation dérivé du langage fonctionnel Lisp, créé dans les années 1990 et basé sur le langage Scheme. Ce langage est le plus adapté pour gérer des listes, chaines de caractères, symboles. Ceci correspond à la conception d'un programme de recherche de preuve.

Overleaf[6] pour écrire ce rapport en Latex.

Avec le confinement il a fallu trouver un moyen de communiquer efficacement pour pouvoir continuer à travailler ensemble. Pour ce faire nous avons utilisé,

Discord[1] pour travailler ensemble, pouvoir faire des partages d'écran mais également pouvoir continuer à faire des réunions avec notre encadrant de projet.

Messenger[5] (la messagerie de Facebook) pour se tenir informés de l'avancée et pouvoir discuter par message plus facilement et rapidement.

Gantt[2] pour organiser le projet et répartir les tâches.

Chapitre 3

Recherche de preuves de la logique propositionnelle

3.1 Usage des recherches de preuves

Les recherches de preuves sont utilisées dans plusieurs domaines mais notamment dans le secteur du Génie Logiciel et de l'Intelligence Artificielle. [7]

Génie Logiciel [3] : Le génie logiciel touche au cycle de vie des logiciels. Toutes les phases de la création d'un logiciel informatique y sont enseignées : l'analyse du besoin, l'élaboration des spécifications, la conceptualisation du mécanisme interne au logiciel ainsi que les techniques de programmation, le développement, la phase de test et finalement la maintenance.

Intelligence Artificielle[4] : l'intelligence artificielle simule des comportements humains (vision, langage, raisonnement) en lien avec les sciences cognitives , elle fait appel à la neurobiologie computationnelle (particulièrement aux réseaux neuronaux), à la logique mathématique (partie des mathématiques et de la philosophie) et à l'informatique. Elle recherche des méthodes de résolution de problèmes à forte complexité logique ou algorithmique. Par extension elle désigne, dans le langage courant, les dispositifs imitant ou remplaçant l'homme dans certaines mises en œuvre de ses fonctions cognitives.

Dans le génie logiciel, la recherche de preuve est utilisée dans la vérification de programme dans plusieurs domaines tels que :

— Transports : avions, trains, automobiles.
— Production d'énergie : contrôle des centrales nucléaires.
— Santé : chaînes de production de médicaments, appareil médicaux.
— Système financier : paiement électronique.
— Militaire

La recherche de preuve permet de contrôler et de vérifier pour éviter des catastrophes tel que l'arrêt de 256 centrales électriques en Amérique du Nord en 2003 causant 11 morts et 6 milliards de dollars de dégâts.

L'intelligence artificielle recherche en permanence des méthodes de résolution de problèmes, ce qui correspond à la recherche de preuve. Elle est basé sur cela, car sans la recherche de preuve une intelligence artificielle ne fonctionnerait pas correctement et il y aurait beaucoup de bugs. Donc sans recherche de preuve, il n'y a pas de méthodes de résolution, et sans méthodes de résolution de problème, il n'y a pas d'intelligence artificielle.

3.2 Calcul des séquents

Un séquent représente une étape d'une démonstration, le calcul des séquents explicitant les opérations possibles sur ce séquent en vue d'obtenir une démonstration complète et correcte. Le calcul des séquents est un système de déduction créé par Gerhard Gentzen. Le nom de ce formalisme fait référence à un style particulier de déduction ; le système original a été adapté à diverses logiques, telles que la logique classique, la logique intuitionniste, la logique modale et la logique linéaire.

- La logique intuitionniste est notamment utilisée pour les programmes fonctionnels certifiés
- La logique modale est utilisée dans l'IA systèmes multiagents, la vérification et bien d'autres domaines.
- La logique linéaire est du point de vue logique, décompose et analyse les logiques classique et intuitionniste et, du point de vue calculatoire, est un système de type pour le lambda-calcul permettant de spécifier certains usages des ressources.

Dans notre projet, le calcul des séquents est le plus adapté. Il permet de trouver plus facilement une solution et la solution est plus lisible et compréhensible. Le calcul des séquents permet également de pouvoir faire des récursions, ce qui correspond à notre langage de programmation est basé sur ceci.

Pour calculer un séquent il faut trouvé la règle adaptée (voir les règles ci-dessous), pour ceci il faut regarder si il y a un connecteurs tel que $A \Rightarrow B$; $A \vee B$; $A \wedge B$; $\neg A$.
Il faut ensuite appliquer la règle qui lui correspond, par exemple :

$$\frac{\overline{A \vdash C, D} \qquad \overline{B \vdash C, D}}{A \vee B \vdash C, D} \vee_{\text{gauche}}$$

Ensuite il y a deux possibilités, soit le séquent ce divise en deux comme dans l'exemple, soit il reste en un seul séquent.

- Dans le cas où il reste en un seul séquent (le plus simple et le plus courant) il suffit de refaire la même chose mais avec la règle adaptée, qui ne sera pas forcément la même, et continuer jusqu'à ne plus avoir de connecteur.

- Dans le cas ou le séquent se sépare en deux, c'est la même chose mais il y a deux fois plus de calculs à faire. Il faut traiter les deux séquents indépendamment et arriver jusqu'à ne plus avoir de connecteur dans le séquent.

Une fois que le séquent n'a plus de connecteur, il faut regarder si il y a un axiome ou non. Pour cela il faut regarder si la même lettre est présente des deux côtés, ce qui signifie alors qu'il y a un axiome, par exemple reprenons le séquent du dessus en le modifier légèrement.

$$\frac{\overline{A \vdash A, D} \qquad \overline{B \vdash A, D}}{A \vee B \vdash A, D} \vee_{\text{gauche}}$$

Dans le séquent de gauche nous pouvons constater qu'il y a un axiome avec A car il est présent des deux côtés.

Malgré qu'il y ait un axiome à gauche, nous en avons pas à droite. Dans ce cas nous pouvons dire que le séquent de base n'est pas vrai pour toutes les valeurs et qu'il y a au moins un contre exemple. Ici nous pouvons donner en contre exemple les valeurs suivantes :

$$\mathrm{B} = 1\,;\ \mathrm{A} = 0\,;\ \mathrm{D} = 0$$

Pour que le séquent de base soit vrai pour toutes valeurs il faut que chaque séquent de fin (sans signe) soit un axiome.

Tout ceci facilite notre algorithme de recherche de preuve en le rendant plus lisible et fonctionnel.

3.3 Règles

$$\frac{}{\Gamma, P \vdash \Delta, P}\ ax$$

$$\frac{}{\Gamma \vdash \Delta, \top}\ \top_{\mathsf{d}}$$

$$\frac{}{\Gamma, \bot \vdash \Delta}\ \bot_{\mathsf{g}}$$

$$\frac{\Gamma \vdash \Delta, P}{\Gamma, \neg P \vdash \Delta}\ \neg_{\mathsf{g}}$$

$$\frac{\Gamma, P \vdash \Delta}{\Gamma \vdash \Delta, \neg P}\ \neg_{\mathsf{d}}$$

$$\frac{\Gamma, P, Q \vdash \Delta}{\Gamma, P \wedge Q \vdash \Delta}\ \wedge_{\mathsf{g}}$$

$$\frac{\Gamma \vdash \Delta, P \qquad \Gamma \vdash \Delta, Q}{\Gamma \vdash \Delta, P \wedge Q}\ \wedge_{\mathsf{d}}$$

$$\frac{\Gamma, P \vdash \Delta \qquad \Gamma, Q \vdash \Delta}{\Gamma, P \vee Q \vdash \Delta}\ \vee_{\mathsf{g}}$$

$$\frac{\Gamma \vdash \Delta, P, Q}{\Gamma \vdash \Delta, P \vee Q}\ \vee_{\mathsf{d}}$$

$$\frac{\Gamma \vdash \Delta, P \qquad \Gamma, Q \vdash \Delta}{\Gamma, P \to Q \vdash \Delta}\ \to_{\mathsf{g}}$$

$$\frac{\Gamma, P \vdash \Delta, Q}{\Gamma \vdash \Delta, P \to Q}\ \to_{\mathsf{d}}$$

Chapitre 4

Conception du programme

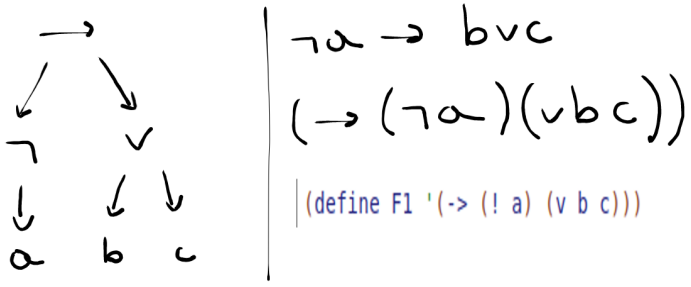
4.1 Représentation des données

Le choix de représentation des données est l'un des plus importants dans la conception de ce programme. Il influe directement sur la façon même dont se pense la résolution du problème. Plusieurs choix sont viables mais certains sont plus avantageux que d'autres.

4.1.1 Représentation d'une formule

La représentation d'une formule se fait de manière préfixée avec les symboles propositionnels servant d'argument à des connecteurs qui prennent le rôle de procédure, de façon similaire à ce qui a été vu en Logique 1. Cette représentation est rendue possible par le fait que toute formule bien formée est un arbre dont les noeuds sont des connecteurs et les feuilles des symboles (voir fig 4.1).

FIGURE 4.1 – Quatre représentations d'une même proposition



4.1.2 Représentation d'un ensemble d'hypothèses/conclusions

Les hypothèses et les conclusions sont représentées de la même manière, soit des listes de formules. Plus précisément, l'ensemble des hypothèses est une liste contenant des formules et de même pour l'ensemble des conclusions.

```
'(A (-> B (^ A C)) E)
```

FIGURE 4.2 – Un exemple d'hypothèses

4.1.3 Représentation de séquents et de preuves

Ces structures étant beaucoup plus complexes, il paraît plus judicieux des les dévoiler lorsqu'on y aura à faire.

4.1.4 Choix d'implémentation des symboles

- NON :! Exemple : '(! A)
- ET : ^ Exemple : '(^A B)
- OU : v Exemple : '(v A B)
- IMPLIQUE : -> Exemple : '(-> A B)

4.2 Jeu de procédures outils

Nous donnons ici la liste des procédures réalisant des instructions élémentaires et qui nous serviront plus tard à implémenter des procédures plus complexes.

- Les procédures de type symbFbf ? (regoupant negFbf ?, andFbf ?, orFbf ? et implFbf ?) prennent une liste et vérifient si le connecteur racine correspond à symb. Si c'est le cas, renvoie vrai, sinon faux.

```
> (andFbf? '(^ A (v B C)))
#t
> (orFbf? '(-> A (^ B C)))
#f
```

FIGURE 4.3 – Appels de symbFbf ?

- La procédure allSymbol ? prend une liste t en argument. Elle vérifie qu'une liste contient exclusivement des symboles propositionnels et pas de connecteurs. Si c'est le cas, renvoie vrai, sinon faux. Si la liste est vide, renvoie faux aussi.

```
> (allSymbol? '(-> A B))
#f
> (allSymbol? '(A B))
#t
```

FIGURE 4.4 – Appels de allSymbol ?

- La procédure firstOp prend une liste t en argument. Elle retourne le premier connecteur qu'elle trouve ou la liste vide si il n'y en a pas.

```
> (firstOp '(A (-> B (^ A C)) E))
'->
> (firstOp '(A B C E))
'()
> (firstOp '((v A (^ B C))(v A B) E))
'v
```

FIGURE 4.5 – Appels de firstOp

— La procédure splitOp prend une liste t en argument et une procédure de type symbFbf ?. Elle retourne la même liste avec en tête la formule qui correspond à symb.

```
> (splitOp '((v A (^ B C))(^ A B) E) andFbf?)
'((^ A B) (v A (^ B C)) E)
> (splitOp '(A (-> A B)) implFbf?)
'((-> A B) A)
```

FIGURE 4.6 – Appels de splitOp

— La procédure allAX prend une liste représentant un arbre de preuve en paramètre et vérifie si toutes ses feuilles sont des axiomes. Si c'est le cas, renvoie true, sinon false.

```
> (allAX '((A B) (C (^ A C)) ∧D ((A B) (A C) AX () ()) ((A B) (C C) NO () ())))
#f
> (allAX '((A B) (A (^ A C)) ∧D ((A B) (A A) AX () ()) ((A B) (C A) AX () ())))
#t
```

FIGURE 4.7 – Appels de allAX

4.3 Implémentation des règles d'inférence

L'implémentation des règles ne se limite pas à donner un résultat correcte sinon aussi utilisable. Par utilisable, on entend formaté de manière à pouvoir l'inclure au sein de l'algorithme de recherche de preuves – et c'est bien là que réside la difficulté. Les différentes procédures comportent par fois de petits détails dans l'ordre de concaténation de listes qui sont en fait nécessaires au bon fonctionnement du programme.

4.3.1 AXIOME

La procédure axiom ? prend deux listes t1 et t2 en argument. Renvoie true à l'unique condition que t1 et t2 soient des listes de symboles sans connecteur et qu'un des symboles de t1 soit aussi dans t2.

L'algorithme est assez simple, si le premier élément de t1 est dans t2 on renvoie true, sinon on rappelle la fonction sur la queue de t1 et t2 en entier.

```
> (axiom? '(A B C) '(D E C))
#t
> (axiom? '(A B C) '(D E F))
#f
```

FIGURE 4.8 – Appels de axiom ?

4.3.2 NON à gauche

La procédure negLeft prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t1 et la procédure negFbf ?. La formule contenant la négation dans t1 est donc placée en tête de t1. On appelle cette nouvelle liste c.
2. On concatène dans l'ordre dans lequel ils sont cités : t2, la tête de la queue de la tête de c et la queue de c.

On se retrouve de cette façon avec une liste dans laquelle on peut discerner ce qui serait le nouvel ensemble d'hypothèses et le nouvel ensemble de conclusions.

```
> (negLeft '(A (! B)) '(C))
'(((A) B C))
> (caar '(((A) B C)))
'(A)
> (cdar '(((A) B C)))
'(B C)
> (cdr '(((A) B C)))
'()
```

FIGURE 4.9 – Appel de negLeft

4.3.3 NON à droite

La procédure negRight prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t2 et la procédure negFbf ?. La formule contenant la négation dans t2 est donc placée en tête de t2. On appelle cette nouvelle liste c.
2. On concatène dans l'ordre dans lequel ils sont cités : la queue de c, t1 et la tête de la queue de la tête de c.

On se retrouve de cette façon avec une liste dans laquelle on peut discerner ce qui serait le nouvel ensemble d'hypothèses et le nouvel ensemble de conclusions.

```
> (negRight '(A) '((! B) (! C)))
'(((B A) (! C)))
> (caar '(((B A) (! C))))
'(B A)
> (cdar '(((B A) (! C))))
'((! C))
> (cdr '(((B A) (! C))))
'()
```

FIGURE 4.10 – Appel de negRight

4.3.4 ET à gauche

La procédure andLeft prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t1 et la procédure andFbf ?. La formule contenant le ET est donc placée en tête de t1. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une liste correctement formatée dans laquelle la formule contenant le symbole du ET a été effacée.

```
> (andLeft '(A (^ C D) E) '(F))
'((C D A E) F)
```

FIGURE 4.11 – Appel de andLeft

4.3.5 ET à droite

La procédure andRight prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t2 et la procédure andFbf ?. La formule contenant le ET est donc placée en tête de t2. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une liste contenant les deux séquents dérivés du premier, la tête contenant le premier, la queue le second.

Aide à la lecture de l'image :

— Dans le premier appel, on récupère les deux séquents produits.
— Dans le second, on isole le premier séquent.
— Dans le troisième, on isole l'ensemble des hypothèses du premier séquent.
— Dans le quatrième, on isole l'ensemble des conclusions du premier séquent.
— Dans le cinquième, on isole le second séquent.
— Dans le sixième, on isole l'ensemble des hypothèses du second séquent.
— Dans le septième, on isole l'ensemble des conclusions du second séquent.

Remarque : On utilisera le même ordre pour les figures suivantes contenant deux séquents.

```
> (andRight '(A) '(B (^ C D)))
'(((A) C B) (A) D B)
> (car '(((A) C B) (A) D B))
'((A) C B)
> (caar '(((A) C B) (A) D B))
'(A)
> (cdar '(((A) C B) (A) D B))
'(C B)
> (cdr '(((A) C B) (A) D B))
'((A) D B)
> (cadr '(((A) C B) (A) D B))
'(A)
> (cddr '(((A) C B) (A) D B))
'(D B)
```

FIGURE 4.12 – Appel de andRight

4.3.6 OU à gauche

La procédure orLeft prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t1 et la procédure orFbf ?. La formule contenant le OU est donc placée en tête de t1. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une liste contenant les deux séquents dérivés du premier, la tête contenant le premier, la queue le second.

```
> (orLeft '(A (v B C)) '(D))
'(((B A) D) (C A) D)
> (car '(((B A) D) (C A) D))
'((B A) D)
> (caar '(((B A) D) (C A) D))
'(B A)
> (cdar '(((B A) D) (C A) D))
'(D)
> (cdr '(((B A) D) (C A) D))
'((C A) D)
> (cadr '(((B A) D) (C A) D))
'(C A)
> (cddr '(((B A) D) (C A) D))
'(D)
```

FIGURE 4.13 – Appel de orLeft

4.3.7 OU à droite

La procédure orRight prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t2 et la procédure orFbf?. La formule contenant le OU est donc placée en tête de t2. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une liste dans laquelle la formule contenant le symbole du OU a été effacée et qui est formatée correctement pour une utilisation dans l'algorithme de recherche de preuve.

```
> (orRight '(A) '(B (v C D) E))
'((A) C D B E)
> (car '((A) C D B E))
'(A)
> (cdr '((A) C D B E))
'(C D B E)
```

FIGURE 4.14 – Appel de orRight

4.3.8 IMPLIQUE à gauche

La procédure impLeft prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t1 et la procédure implFbf?. La formule contenant le –> est donc placée en tête de t1. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une liste contenant les deux séquents dérivés de l'original, la tête contenant le premier, la queue le second.

```
> (impLeft '(A (-> B C)) '(D))
'(((A) B D) (C A) D)
> (car '(((A) B D) (C A) D))
'((A) B D)
> (caar '(((A) B D) (C A) D))
'(A)
> (cdar '(((A) B D) (C A) D))
'(B D)
> (cdr '(((A) B D) (C A) D))
'((C A) D)
> (cadr '(((A) B D) (C A) D))
'(C A)
> (cddr '(((A) B D) (C A) D))
'(D)
```

FIGURE 4.15 – Appel de impLeft

4.3.9 IMPLIQUE à droite

La procédure impRight ? prend deux listes t1 et t2 en argument. Elle renvoie une seule liste qui est transformée de la façon suivante :

1. On applique splitOp sur t2 et la procédure implFbf ?. La formule contenant le –> est donc placée en tête de t2. On appelle cette nouvelle liste c.
2. On concatène de façon un peu plus compliquée cette fois. (Voir code pour le détail)

On se retrouve de cette façon avec une réorganisation des éléments conforme à la règle de l'implique à droite.

```
> (impRight '(A) '(B (-> C D)))
'(((C A) D B))
> (caar '(((C A) D B)))
'(C A)
> (cdar '(((C A) D B)))
'(D B)
```

FIGURE 4.16 – Appel de impRight

4.4 Algorithme de recherche de preuve

L'algorithme de preuve nommé inference prend deux listes H (comme Hypothèses) et C (comme Conclusions) en argument et renvoie une liste représentant un arbre de preuve avec les règles utilisées à chaque noeud. Niveau structure, ce sont quatre conditions qui sont testées dès le début du programme. On consacre sa partie à chacune d'elles.

4.4.1 H |– C est un axiome

On vérifie grâce à axiom ? si le séquent courant est un axiome. Si tel est le cas, on est arrivé à l'une des feuilles de l'arbre de preuve. On représente le nouveau séquent en concaténant dans cette ordre t1, t2, le mot AX symbolisant la règle utilisée suivi de deux listes vides '() '() qui servent à faciliter l'affichage de la preuve. Ces listes vident signifient qu'aucun nouveau séquent n'est produit à partir de cette règle.

```
> (inference '(A B) '(B))
'((A B) (B) AX () ())
```

FIGURE 4.17 – Appel de inference sur un axiome

4.4.2 H non vide et non uniquement composé de symboles

On vérifie si H non nul et H non uniquement symbolique sont tous les deux vrais. Si c'est le cas on récupère le premier connecteur de H. On entre alors dans un nouvel embranchement.

Le premier connecteur est une négation

Dans ce cas, on doit appliquer le NON à gauche. On déclare pour cela deux variables K et L.

K est le résultat de la règle du NON à gauche appliquée à H et C.

L est le rappel récursif de l'inférence sur le caar de K (représentant en fait le nouveau H formaté proprement) et sur le cdar de K (représentant le nouveau C). Pour plus de détails sur l'isolation des différents éléments des séquents produits, veuillez remonter sur les figures associées à chaque règle dans la partie 4.3.

On concatène ensuite dans l'ordre t1, t2, !G représentant la règle utilisée, L et la liste vide (puisqu'un seul nouveau séquent est produit).

```
> (inference '((! A) B) '(B))
'(((! A) B) (B) ¬G ((B) (A B) AX () ()) ())
```

FIGURE 4.18 – Appel de inference sur une négation à gauche

Le premier connecteur est une conjonction

Dans ce cas, on doit appliquer le ET à gauche. On déclare pour cela deux variables K et L.
K est le résultat de la règle du ET à gauche appliquée à H et C.
L est le rappel récursif de l'inférence sur le car de K (représentant le nouveau H) et le cdr de K (représentant le nouveau C).
On concatène ensuite t1, t2, ^G, L et la liste vide (puisqu'un seul nouveau séquent est produit).

```
> (inference '((^ A B)) '(B))
'(((^ A B)) (B) ∧G ((A B) (B) AX () ()) ())
> (inference '((^ A B)) '(C))
'(((^ A B)) (C) ∧G ((A B) (C) NO () ()) ())
```

FIGURE 4.19 – Appel de inference sur un ET à gauche

Le premier connecteur est une disjonction

Dans ce cas, on doit appliquer le OU à gauche. On déclare pour cela trois variables K, L, R.
K est le résultat de la règle du OU à gauche appliquée à H et C.
L est le rappel récursif de l'inférence sur le caar de K (représentant le nouveau H1) et le cdar de K (représentant le nouveau C1).
R est le rappel récursif de l'inférence sur le cadr de K (représentant le nouveau H2) et le cddr de K (représentant le nouveau C2).

On concatène ensuite t1,t2, vG, L et R (et pas de liste vide, L et R étant les deux nouveaux séquents produits).

```
> (inference '((v A B)) '(A))
'(((v A B)) (A) vG ((A) (A) AX () ()) ((B) (A) NO () ()))
```

FIGURE 4.20 – Appel de inference sur un OU à gauche

Le premier connecteur est une implication

Dans ce cas, on doit appliquer le IMPLIQUE à gauche.On déclare pour cela trois variables K, L, R.

K est le résultat de la règle du implique à gauche appliquée à H et C.

L est le rappel récursif de l'inférence sur le caar de K (représentant le nouveau H1) et le cdar de K (représentant le nouveau C1).

R est le rappel récursif de l'inférence sur le cadr de K (représentant le nouveau H2) et le cddr de K (représentant le nouveau C2).

On concatène ensuite t1,t2, –>G, L et R (et pas de liste vide, L et R étant les deux nouveaux séquents produits).

```
> (inference '(A (-> B C)) '(C))
'((A (-> B C)) (C) ->G ((A) (B C) NO () ()) ((C A) (C) AX () ()))
```

FIGURE 4.21 – Appel de inference sur un –> à gauche

4.4.3 C non vide et non uniquement composé de symboles

On vérifie si C non nul et C non uniquement symbolique sont tous les deux vrais. Si c'est le cas on récupère le premier connecteur de C. On entre alors dans un nouvel embranchement.

Le premier connecteur est une négation

Dans ce cas, on doit appliquer le NON à droite. On déclare pour cela deux variables K et L.

K est le résultat de la règle du NON à droite appliquée à H et C.

L est le rappel récursif de l'inférence sur le caar de K (représentant en fait le nouveau H formaté proprement) et sur le cdar de K (représentant le nouveau C).

Pour plus de détails sur l'isolation des différents éléments des séquents produits, veuillez remonter sur les figures associées à chaque règle dans la partie 4.3.

On concatène ensuite dans l'ordre $t1, t2, !D$ représentant la règle utilisée, L et la liste vide (puisqu'un seul nouveau séquent est produit).

```
> (inference '(A) '(B (! B)))
'((A) (B (! B)) ¬D ((B A) (B) AX () ()) ())
```

FIGURE 4.22 – Appel de inference sur une négation à droite

Le premier connecteur est une conjonction

Dans ce cas, on doit appliquer le ET à droite. On déclare pour cela trois variables K, L, R.

K est le résultat de la règle du ET à droite appliquée à H et C.

L est le rappel récursif de l'inférence sur le caar de K (représentant le nouveau H1) et le cdar de K (représentant le nouveau C1).

R est le rappel récursif de l'inférence sur le cadr de K (représentant le nouveau H2) et le cddr de K (représentant le nouveau C2).

On concatène ensuite t1,t2, ^D, L et R (et pas de liste vide, L et R étant les deux nouveaux séquents produits).

```
> (inference '(A) '(B (^ A C)))
'((A) (B (^ A C)) ∧D ((A) (A B) AX () ()) ((A) (C B) NO () ()))
```

FIGURE 4.23 – Appel de inference sur un ET à droite

Le premier connecteur est une disjonction

- Dans ce cas, on doit appliquer le OU à droite. On déclare pour cela deux variables K et L. K est le résultat de la règle du OU à droite appliquée à H et C.
- L est le rappel récursif de l'inférence sur le car de K (représentant le nouveau H) et le cdr de K (représentant le nouveau C).
- On concatène ensuite t1, t2, vD, L et la liste vide (puisqu'un seul nouveau séquent est produit).

```
> (inference '(A) '(B (v B C)))
'((A) (B (v B C)) vD ((A) (B C B) NO () ()) ())
> (inference '(A) '(B (v A B)))
'((A) (B (v A B)) vD ((A) (A B B) AX () ()) ())
```

FIGURE 4.24 – Appel de inference sur un OU à droite

Le premier connecteur est une implication

Dans ce cas, on doit appliquer le IMPLIQUE à droite. On déclare pour cela deux variables K et L.

K est le résultat de la règle du IMPLIQUE à droite appliquée à H et C.

L est le rappel récursif de l'inférence sur le caar de K (représentant le nouveau H) et le cdar de K (représentant le nouveau C).

On concatène ensuite t1, t2, –>D, L et la liste vide (puisqu'un seul nouveau séquent est produit).

```
> (inference '(A) '((-> B C) C))
'((A) ((-> B C) C) ->D ((B A) (C C) NO () ()) ())
> (inference '(A) '((-> B A) C))
'((A) ((-> B A) C) ->D ((B A) (A C) AX () ()) ())
```

FIGURE 4.25 – Appel de inference sur un –> à droite

4.4.4 La branche de sortie

Si aucune des trois conditions précédentes n'est satisfiable, on renvoie une simple concaténation de t1, t2 'NO et deux listes vides.

```
> (inference '(A) '(B))
'((A) (B) NO () ())
```

FIGURE 4.26 – Appel de inference sur un séquent non valide

4.5 Le système de listes vides

On remarque que selon les règles, on concatène une, deux ou aucune liste vide à t1, t2 ainsi que le nom de la règle appliquée. Ces listes vides nous permettent en fait de représenter les séquents dérivés de l'original. La règle du ET à droite produit par exemple deux nouveaux séquents qu'on a nommé L et R (pour nouveau séquent gauche et nouveau séquent droit), on ne rajoute donc pas de liste vide. A contrario, la règle de l'axiome ne produit aucun nouveau séquent, on concatène donc deux listes vides à la fin de la règle. Cela permet tout simplement de créer une standardisation de la longueur des règles afin de pouvoir programmer une règle générale d'affichage.

4.6 Un exemple de preuve

Voyons à quoi ressemble la résolution du séquent suivant avec l'algorithme :

$$\overline{J \implies \neg\neg I, (H \vee G) \implies J, F \implies H, G \wedge F \vdash I}$$

On comprend directement l'importance d'avoir un algorithme d'affichage qui permette de rendre la preuve lisible.

```
> (inference '((-> J (! (! I))) (-> (v H G) J) (-> F H) (^ G F)) ↵
'(I))
'(((-> J (! (! I))) (-> (v H G) J) (-> F H) (^ G F))
  (I)
  ->G
  (((-> (v H G) J) (-> F H) (^ G F))
   (J I)
   ->G
   (((-> F H) (^ G F))
    ((v H G) J I)
    ->G
    (((^ G F)) (F (v H G) J I) ∧G ((G F) (F (v H G) J I) ∨D ((G  ↵
F) (H G F J I) AX () ()) ()) ())
    ((H (^ G F)) ((v H G) J I) ∧G ((G F H) ((v H G) J I) ∨D ((G  ↵
F H) (H G J I) AX () ()) ()) ()))
   ((J (-> F H) (^ G F)) (J I) ->G ((J (^ G F)) (F J I) ∧G ((G F ↵
J) (F J I) AX () ()) ()) ((H J (^ G F)) (J I) ∧G ((G F H J) (J  ↵
I) AX () ()) ())))
  (((! (! I)) (-> (v H G) J) (-> F H) (^ G F))
   (I)
   ¬G
   (((-> (v H G) J) (-> F H) (^ G F))
    ((! I) I)
    ->G
    (((-> F H) (^ G F))
     ((v H G) (! I) I)
     ->G
     (((^ G F)) (F (v H G) (! I) I) ∧G ((G F) (F (v H G) (! I)  ↵
I) ∨D ((G F) (H G F (! I) I) ¬D ((I G F) (H G F I) AX () ()) ()) ↵
()) ())
     ((H (^ G F)) ((v H G) (! I) I) ∧G ((G F H) ((v H G) (! I)  ↵
I) ∨D ((G F H) (H G (! I) I) ¬D ((I G F H) (H G I) AX () ()) ()) ↵
()) ()))
    ((J (-> F H) (^ G F))
     ((! I) I)
     ->G
     ((J (^ G F)) (F (! I) I) ∧G ((G F J) (F (! I) I) ¬D ((I G F ↵
J) (F I) AX () ()) ()) ())
     ((H J (^ G F)) ((! I) I) ∧G ((G F H J) ((! I) I) ¬D ((I G F ↵
H J) (I) AX () ()) ()) ()))
   ()))
```

FIGURE 4.27 – Appel de inference sur un séquent

4.7 Interface graphique

La lisibilité d'une preuve non formatée étant très limitée, il nous semblait indispensable de concevoir sur mesure une méthode d'affichage permettant de représenter la preuve de manière lisible afin de donner une utilité à l'algorithme de recherche. Par faute de temps, nous avons opté pour une représentation directement sur DrRacket (l'environnement utilisé pour programmer en Racket). Par soucis de clarté, on ne rentrera cependant pas dans les détails de l'implémentation des différentes fonctions, on se contentera ici de dire ce qu'elles font. (code source disponible dans l'archive pour plus de détails.)

Les procédures sont au nombre de quatre : trois fonctions outils et une fonction directrice. Elles sont basées sur ce qui s'appelle en Racket le séquencement grâce à la structure begin permettant d'évaluer des expressions dans l'ordre.

Parmi les fonctions outils, une s'occupe de l'indentation verticale de la preuve, une s'occupe de l'indentation horizontale d'une branche qui n'est pas la dernière de sa profondeur et une qui s'occupe de l'indentation d'une branche qui est la dernière de sa profondeur.

4.7.1 Un exemple de preuve

Voici ce que donne l'affichage de la preuve du séquent de la figure 4.27 :

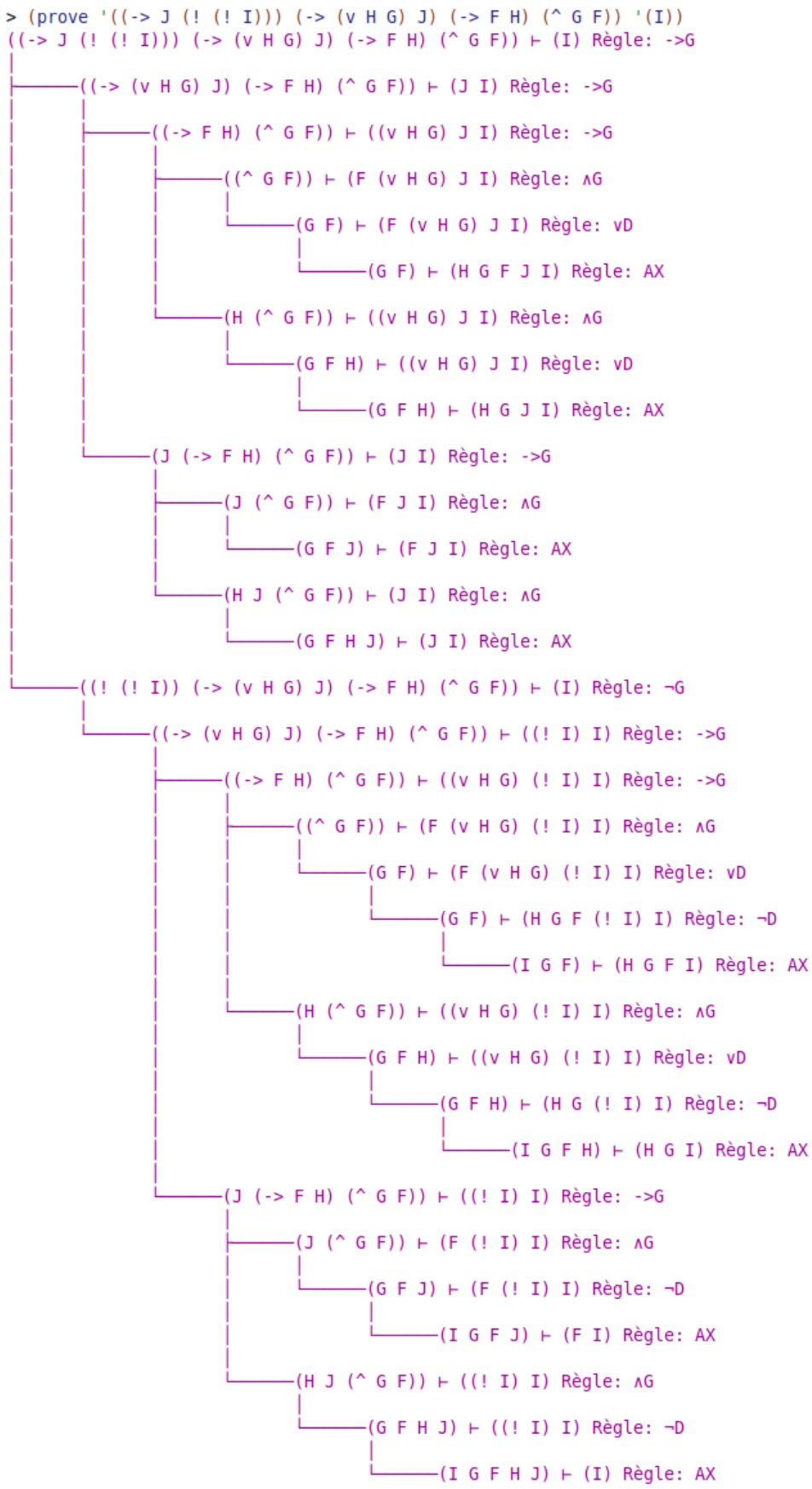


FIGURE 4.28 – Preuve sur un séquent valide

Dans le cas d'un séquent non valide, on renvoie aussi un contre-modèle :

```
> (prove'((^ (v A B) C)) '((-> C A)))
((^ (v A B) C)) ⊢ ((-> C A)) Règle: ∧G
│
└──────((v A B) C) ⊢ ((-> C A)) Règle: ∨G
       │
       ├──────(A C) ⊢ ((-> C A)) Règle: ->D
       │      │
       │      └──────(C A C) ⊢ (A) Règle: AX
       │
       └──────(B C) ⊢ ((-> C A)) Règle: ->D
              │
              └──────(C B C) ⊢ (A) Règle: NO
'((B . 1) (A . 0) (C . 1))
```

FIGURE 4.29 – Appel de inference sur un séquent non valide

4.8 Résolution du problème posé en intro

On s'attachera ici à la résolution du problème posé en introduction. Pour rappel :

S'il est venu Seul, il a pris le Bus ou le Train. S'il a pris le Bus ou son Automobile alors il est arrivé en Retard et il a Manque la réunion. Il n'est pas arrivé en Retard.
Peut-on en conclure que « s'il est venu Seul, il a pris le Train » ?

Il nous faut modéliser le problème sous forme de séquent.
Modélisons dans un premier temps les symboles élémentaires :

- S = Il est venu Seul
- B = Il a pris le Bus
- T = Il a pris le Train
- A = Il a pris son Automobile
- R = Il est arrivé en Retard
- M = Il a Manqué la réunion

Modélisons maintenant les hypothèses :
- H1 = *S'il est venu Seul, il a pris le Bus ou le Train.* = S -> B v T
- H2 = *S'il a pris le Bus ou son Automobile alors il est arrivé en Retard et il a Manque la réunion* = (B v A) -> (R^M)
- H3 = *Il n'est pas arrivé en Retard.* = ! R

Modélisons maintenant les conclusions :
- C1 = *S'il est venu Seul, il a pris le Train* = S -> T

On veut donc prouver :

$$\frac{}{H1, H2, H3 \vdash C1}$$

L'algorithme trouve bien une preuve confirmant que la conclusion est valide avec nos hypothèses.

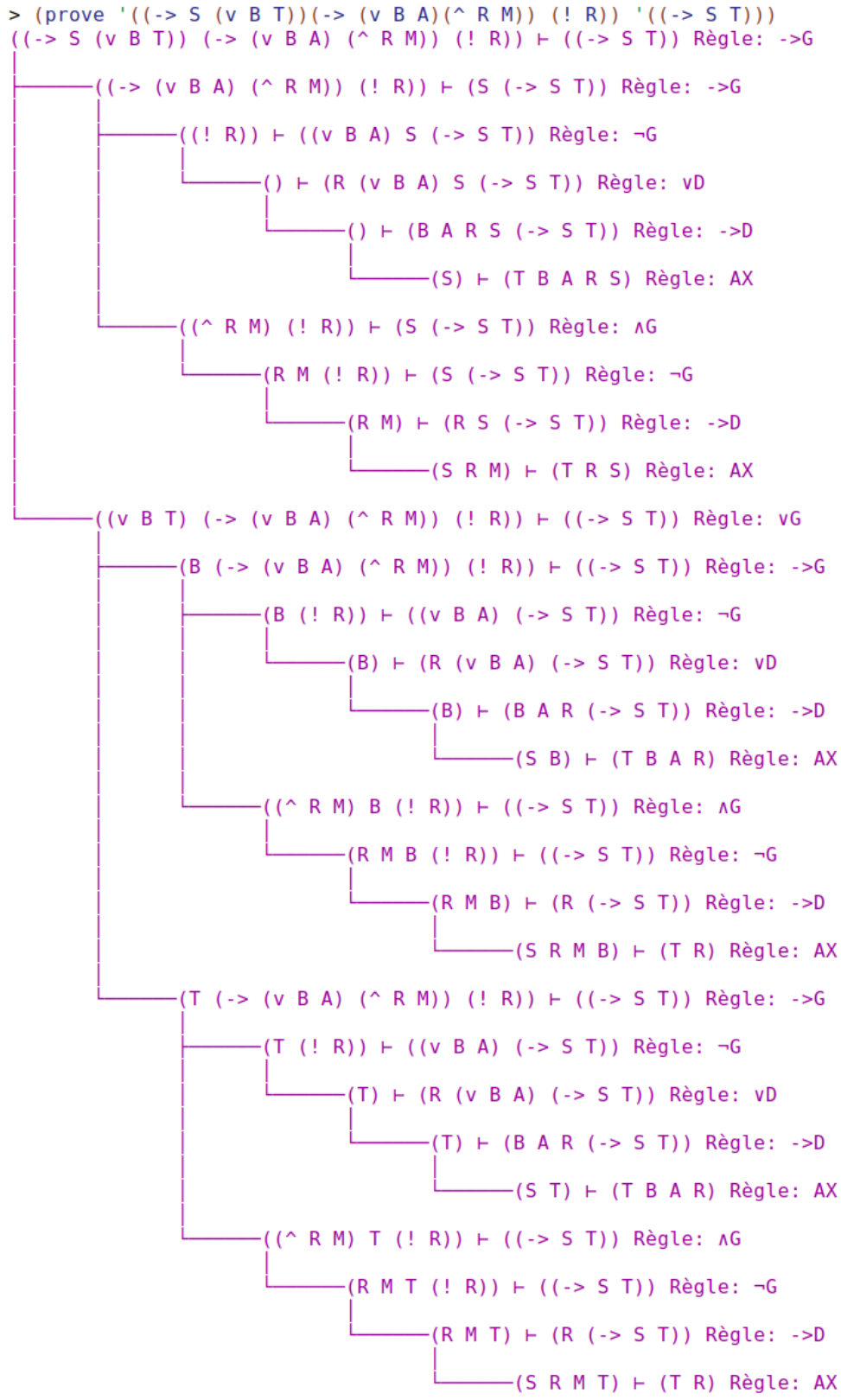


FIGURE 4.30 – Preuve du problème

Chapitre 5

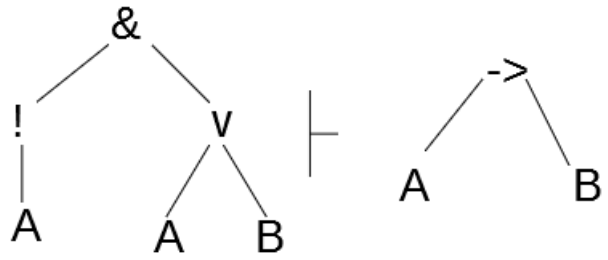
Bilan et difficultés rencontrées

5.1 Bilan du projet

Nous avons fini le programme de recherche de preuve en y implémentant toutes les règles comme vu dans les parties précédentes. Ce programme, ainsi que la maîtrise des nombreuses règles et de comment tout cela fonctionne, nous a pris beaucoup de temps et nous avons du revoir nos souhait d'interfaces.

Nous avons malgré tout décidé de conserver l'idée de faire une interface de sortie pour rendre le rendu plus lisible et agréable à lire. Vous avez pu voir le rendu dans la partie 4. Nous ne voulions pas prendre une interface de sortie déjà faite mais la faire par nous-même pour que le rendu du programme nous correspondes plus et soit plus "authentique".

Nous n'avons pas eu le temps de faire une interface d'entrée, mais nous aurions aimé faire une interface d'entrée en forme d'arbre, en choisissant le connecteur ou la variable à chaque étape de l'arbre (un schéma ci-dessous). Ceci nous aurait permis de pouvoir rentrer le séquent de base plus simplement et sans passer par un appel du programme. Cela aurait été lisible et simple à utiliser pour n'importe qui.



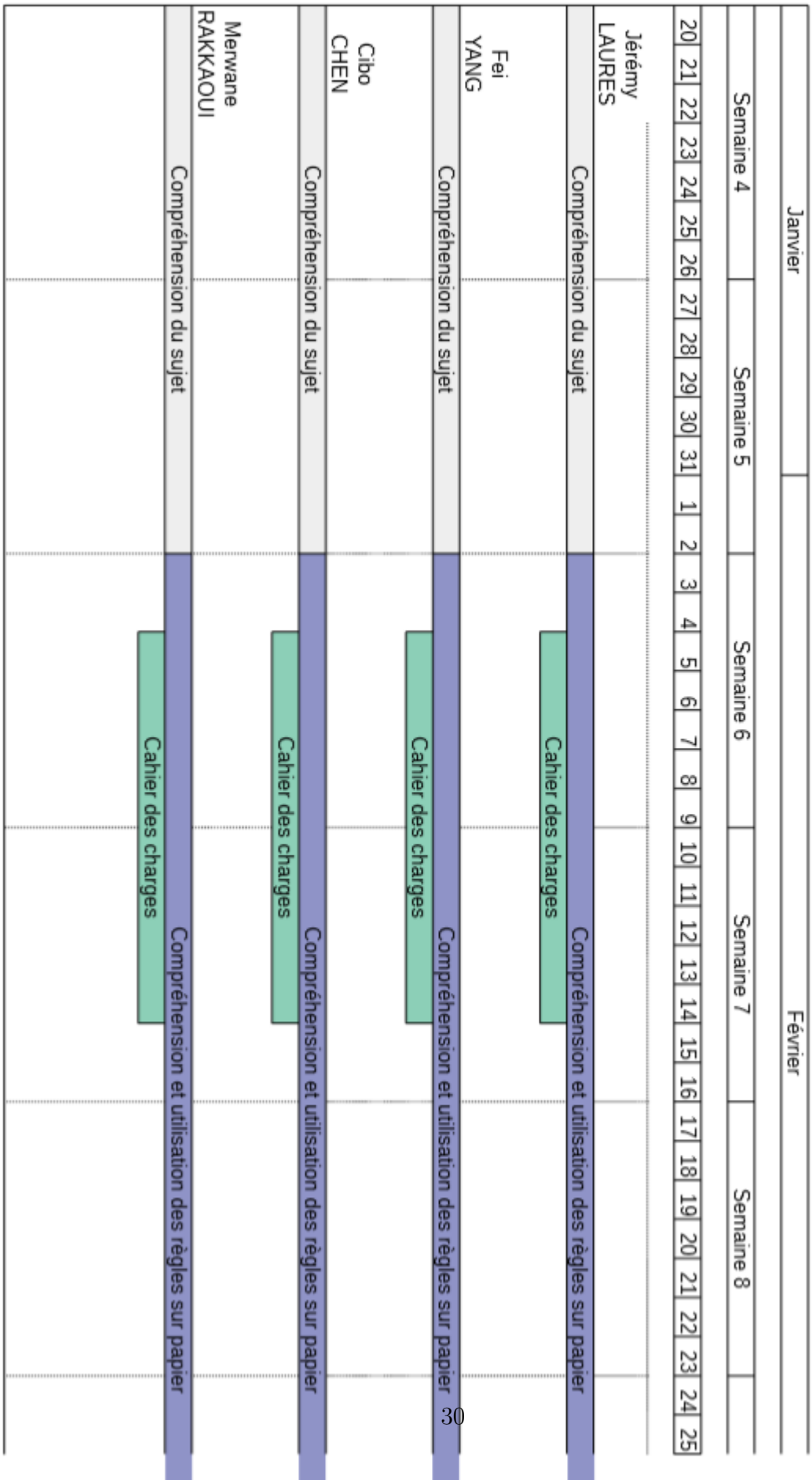
5.2 Difficultés rencontrées

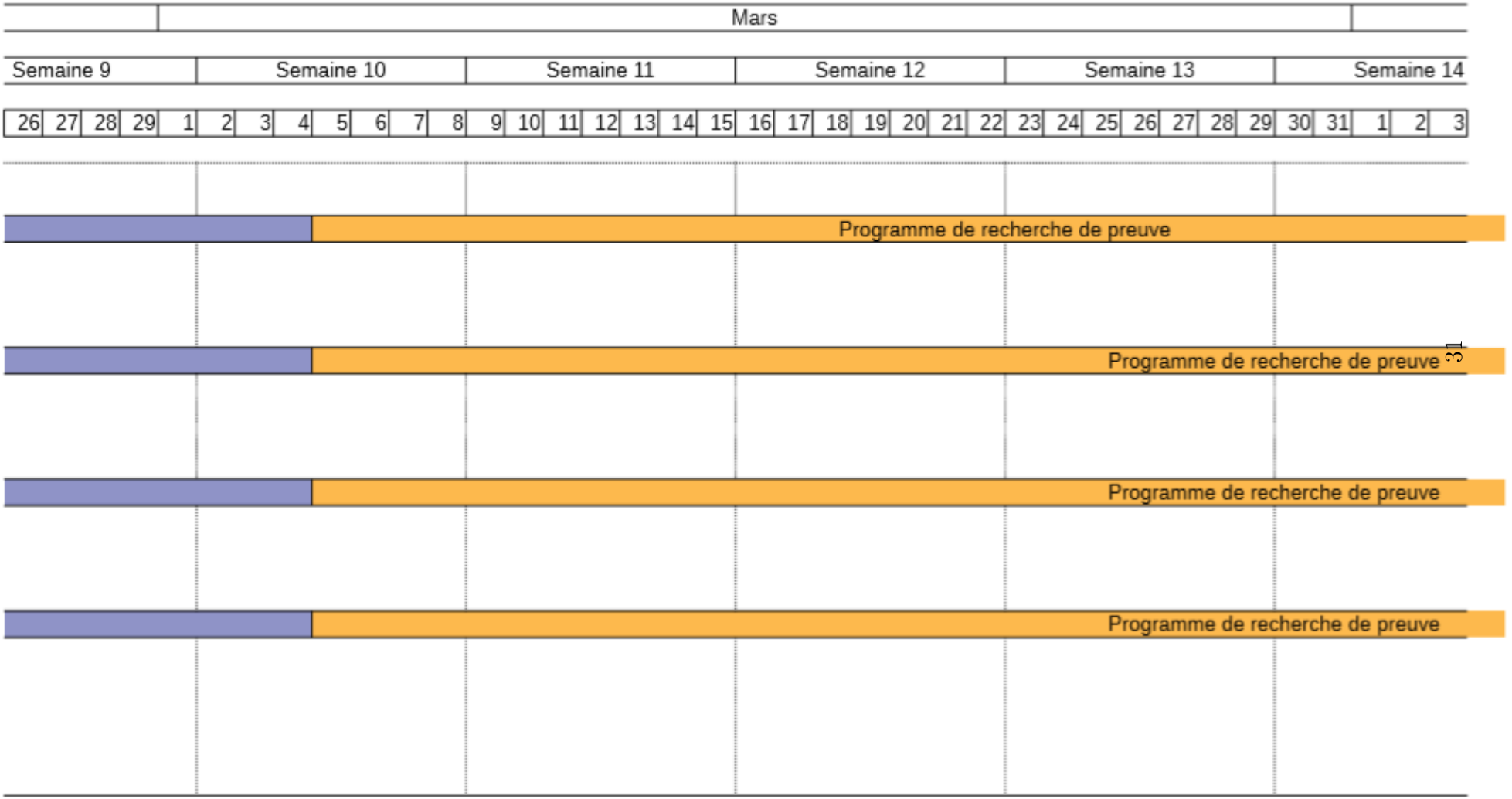
Nous avons rencontré des difficultés lors de la programmation de l'interface graphique puisque cela est nouveau pour nous et que nous avons du le faire rapidement.
Nous avons également eu du mal à afficher les listes normalement et non en préfixe, ainsi que de programmer les règles qui donnent deux séquents. Tout ceci était nouveau pour nous mais nous avons réussi à passer par dessus ces difficultés

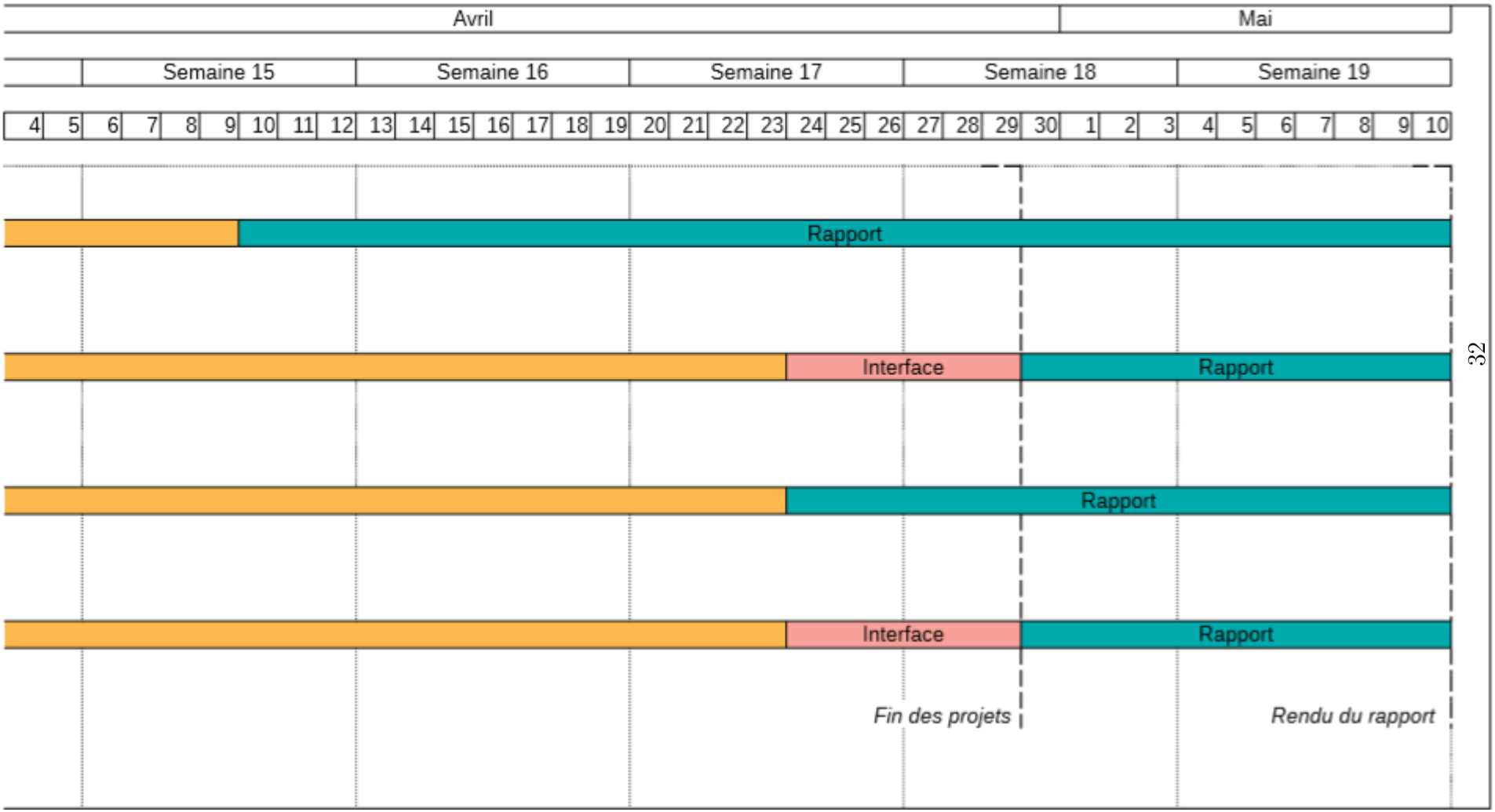
Chapitre 6

Annexes

6.1 Diagramme de Gantt




Mars
Semaine 9
Semaine 10
Semaine 11
Semaine 12
Semaine 13
Semaine 14
26 27 28 29 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3
Programme de recherche de preuve
Programme de recherche de preuve
Programme de recherche de preuve
Programme de recherche de preuve


Avril
Mai
Semaine 15
Semaine 16
Semaine 17
Semaine 18
Semaine 19
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 1 2 3 4 5 6 7 8 9 10
Rapport
Interface
Rapport
Rapport
Interface
Rapport
Fin des projets
Rendu du rapport

Bibliographie

[1] Discord. https://discord.com/.

[2] Gantt. https://www.gantt.com/fr/.

[3] Génie logiciel. https://fr.wikipedia.org/wiki/GÃľnie_logiciel.

[4] Intelligence artificielle. https://fr.wikipedia.org/wiki/Intelligence_artificielle.

[5] Messenger. https://www.messenger.com/.

[6] Overleaf. https://fr.overleaf.com/.

[7] Jean H. Gallier. *Logic for computer science*. Harper & Row Publishers, New York, 1986.