Formalisation des démonstrations pour l'étude des possibilités informatiques d'assistant de preuves.

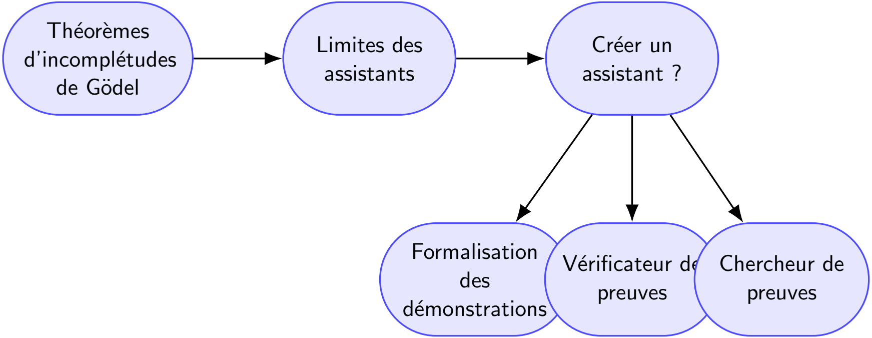
Florian DEVARENNE
n°45154

Introduction

Kurt Gödel

Gerhard Gentzen

Origine de ma réflexion



Problématique et Ancrage au thème

Problématique

Comment formaliser et implémenter un moteur de vérification de preuve en **déduction naturelle** en `OCaml`, tout en facilitant la compréhension des preuves grâce à une **traduction en langage naturel** ?

Lien avec le thème

Transition, **Transformation**, **Conversion** :

- Formalisation d'une preuve
- Transformation d'un arbre de preuve en preuve en français (traducteur)

Sommaire

Préliminaires : Déduction naturelle et Choix d'implémentation

La déduction naturelle

Déduction naturelle

- système formalisant les démonstrations
- reposant sur des règles de déduction

Pour mon projet en particulier

- Calcul propositionnel (pas de $\forall, \exists$)

$$\dfrac{\dfrac{\text{il pleut} \qquad \text{il pleut} \to \text{sol mouillé}}{\text{sol mouillé}} \qquad \text{sol mouillé} \to \text{sol glissant}}{\text{sol glissant}}$$

Une règle et un séquent en détail

- Une règle d'élimination et d'introduction pour $\wedge, \vee, \neg, \rightarrow$
- Une règle pour l'absurdité et l'affaiblissement

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \,(\rightarrow \mathrm{i})$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash A \rightarrow B}{\Gamma \vdash B} \,(\rightarrow \mathrm{e})$$

Un séquent :

$$\Gamma \vdash A \rightarrow B$$

un exemple de preuve courte

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A, A \vdash A}\,(\mathrm{Ax}) \qquad \dfrac{}{\neg A, A \vdash \neg A}\,(\mathrm{Ax})}{\neg A, A \vdash \bot}\,(\neg\ \mathrm{e})}{\neg A \vdash (A \to \bot)}\,(\to\ \mathrm{i})}{\vdash\ \neg A \to (A \to \bot)}\,(\to\ \mathrm{i})$$

Choix d'implémentation

- Type enregistrement/structuré pour sequent et regle
- Type somme pour Formule
- Structure d'arbre pour demonstration

```
type formule =
  |Var of char
  |Bot
  |Top
  |Impl of formule*formule
  |Neg of formule
  |Conj of formule*formule
  |Disj of formule*formule;;

type sequent =
  {
    hypotheses : formule list;
    objectif : formule;
  };;

type regle  =
  {
    premisses : sequent list;
    conclusion : sequent;
  };;

type 'a arbredemo =
  |N of regle * 'a arbredemo list;;

type demonstration = regle arbredemo;;
```

Objectif 1: Vérificateur de preuve

Première objectif : vérificateur de preuve

On veut construire une fonction OCaml de signature demonstration $\rightarrow$ *bool*

Pour y arriver :

- Quelle règle doit respecter une preuve ?
- Vérifier ces points facilement en fonction de nos choix d'implémentation

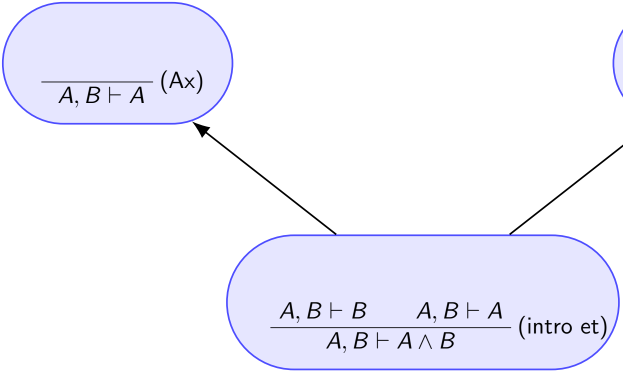
Principe de fonctionnement

Vérification de validité

- Pour chaque lien entre les règles : conclusion de la règle du dessus correspond à l'une des prémisses de la règle d'en dessous
- Les feuilles de l'arbre sont des "Axiomes"
- Précondition : utilisation des règles implémentées uniquement

```
let rec est_valide (d : demonstration) : bool = match d with
  |N(r,[])->  (List.mem r.conclusion.objectif r.conclusion.hypotheses)
  |N(r,l) -> (jonction_correcte r l) && (list_all_true est_valide l)
;;
```

Visualisation de l'arbre de preuve



Test de notre verificateur sur nos exemples

Vérification de la preuve des séquents :

- A, B $\vdash A \wedge B$
- $\vdash \ \neg A \rightarrow (A \rightarrow \bot)$

Résultat des tests

```
# est_valide demo_presentation;;
- : bool = true
# est_valide demo1;;
- : bool = true
```

Objectif 2: Traducteur/Assistant de preuve

Deuxième objectif : traducteur/assistant de preuve

On veut construire une fonction OCaml de signature
sequent $\rightarrow$ bool

Pour y arriver :

- aide l'utilisateur à créer une preuve étape par étape
- traduction en français des regles
- implémenter afin de facilité la vérification par notre programme précédent

Principe de fonctionnement

Découpage en étape

- Fonction construction_demo : sequent $\rightarrow$ demonstration
- Fonction assistant ci-dessous

```
let assistant (s : sequent) : bool =
  print_endline "L'assistant va vous aider à montrer le séquent : ";
  print_sequent s;
  print_endline "Puis va vous indiquez si votre preuve est correcte";
  let d = construction_demo s in
  print_endline "Votre démonstration est terminé. Demandons à notre prog
  est_valide d
;;
```

Fonctionnement pratique

```
florian@florian-ThinkPad-X13:~/Documents/mp2i-mpi/info/5/TIPE/code$ ./tipe
L'assistant va vous aider à montrer le séquent : A , B ,  ⊢A et B
Puis va vous indiquez si votre preuve est correcte
On veut maintenant montrer le séquent : A , B ,  ⊢A et B
Voici les prochaines phrases que vous pouvez utilisez :
Phrases 1  : On veut montrer X -> Y en supposant L ,  _.
 Supposons L ,  _ et X montrons Y
Phrases 2  : On veut montrer Y en supposant L ,  _.
 Or L ,  _ permet de monter X -> Y et X on a donc Y
```

Test sur exemple

On montre le séquent $A, B \vdash A \wedge B$

- On commence par choisir la règle $\wedge i$
- Le programme nous demande donc de prouver $A, B \vdash A$ puis $A, B \vdash B$
- On prouve ces deux séquents avec axiome
- Fin, on obtient le résultat :

```
Votre démonstration est terminé. Demandons à notre programme si elle est correcte
Le résultat est : true
```

Test sur exemple

La preuve en français donné par notre programme

On veut montrer $A \wedge B$ en supposant B , A or B , A permet individuellement de montrer A et aussi B donc on a $A \wedge B$
On veut montrer A en supposant B , A puis ce qu'on le suppose, on a bien A
On veut montrer B en supposant B , A puis ce qu'on le suppose, on a bien B

Conclusion et Ouverture

Les assistants de nos jours

- Coq, Isabelle , Lean
- Méthode complexe : Tableaux, Unifications
- IA ?
- Utilisation

Retours et Critiques

Points positifs

- Résultats satisfaisants
- Fonctionne pour tout séquent de calcul propositionnel

Axes d'amélioration

- Meilleure interface utilisateur
- Aide pour la preuve autre que traduction
- Implémentation moins lourde

Conclusion

Pour rêver à l'infini des nombres, il faudra toujours des mathématiciennes et des mathématiciens.

- Arte, Voyage au pays des maths , L' Entscheidungsproblem ou la fin des mathématiques ?

Annexe : code et tests

Règle p1

Axiomes

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{ax} \qquad \frac{}{\Gamma \vdash \top}\ \top$$

Règles d'introduction

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to\mathsf{i} \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\mathsf{i} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}\ \wedge\mathsf{i}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\mathsf{i\text{-}g} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash B \vee A}\ \vee\mathsf{i\text{-}d}$$

Règle p2

$$\frac{\Gamma \vdash A \qquad \Delta \vdash A \to B}{\Gamma, \Delta \vdash B} \to\text{e} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash \neg A}{\Gamma, \Delta \vdash \bot} \neg\text{e}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{e-g} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{e-d}$$

$$\frac{\Gamma \vdash A \vee B \qquad \Delta, A \vdash C \qquad \Delta', B \vdash C}{\Gamma, \Delta, \Delta' \vdash C} \vee\text{e}$$

Règles pour l'absurde

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A} \text{abs}$$

Test de l'assistant p1

```
florian@florian-ThinkPad-X13:~/Documents/mp2i-mpi/info/5/TIPE/code$ ./tipe
L'assistant va vous aider à montrer le séquent : A , B ,   ⊢A et B
Puis va vous indiquez si votre preuve est correcte
On veut maintenant montrer le séquent : A , B ,   ⊢A et B
Voici les prochaines phrases que vous pouvez utilisez :
Phrases 1  : On veut montrer X -> Y en supposant L ,  _.
 Supposons L ,  _ et X montrons Y
Phrases 2  : On veut montrer Y en supposant L ,  _.
 Or L ,  _ permet de monter X -> Y et X on a donc Y
Phrases 3  : On veut montrer X en supposant L ,  _ et Y
 or L ,  _ permet de montrer X donc on a X (sans supposer Y)
Phrases 4  : On veut montrer X ∧ Y en supposant L ,  _
 or L ,  _ permet individuellement de montrer X et aussi Y donc on a X ∧ Y
Phrases 5  : On veut montrer Y en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a Y
Phrases 6  : On veut montrer X en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a X
Phrases 7  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer Y donc on a X ∨ Y
Phrases 8  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer X donc on a X ∨ Y
Phrases 9  : On veut montrer Z en supposant L ,  _
 or L ,  _ permet de montrer X ∨ Y
et en supposant L ,  _ , X on peut montrer Z
et en supposant L ,  _ , X on peut montrer Z
 on peut donc finalement montrer Z
Phrases 10  : On veut montrer ¬X en supposant L ,  _
 or on peut montrer l'absurde (Bot) en supposant X , L ,  _ donc on a ¬X
Phrases 11  : On veut montrer que L ,  _ est absude
 or on peut montrer X et ¬X en supposant L ,  _ donc L ,  _ est absurde
Phrases 12  : On veut montrer X en supposant L ,  _
 or on peut montrer l'absurde (Bot) en supposant L ,  _ et ¬X donc on a X
Phrases 13  : On veut montrer X en supposant L ,  _
 puis ce qu'on le suppose, on a bien X
Tapez le numéro de la phrase que vous voulez écrire :
4
```

Test de l'assistant p2

```
Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qu
i remplace le L) :
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
1
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : A
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
1
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : B
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
0
Rentrez maintenant la formule que vous voulez pour X :
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : A
```

Test de l'assistant p3

```
Rentrez maintenant la formule que vous voulez pour Y :
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : B
On veut montrer A ∧ B en supposant B ,  A ,  _
 or B ,  A ,  _ permet individuellement de montrer A et aussi B donc on a A ∧ B
On veut maintenant montrer le séquent : B , A ,   ⊢ A
Voici les prochaines phrases que vous pouvez utilisez :
Phrases 1  : On veut montrer X -> Y en supposant L ,  _.
 Supposons L ,  _ et X montrons Y
Phrases 2  : On veut montrer Y en supposant L ,  _.
 Or L ,  _ permet de monter X -> Y et X on a donc Y
Phrases 3  : On veut montrer X en supposant L ,  _ et Y
 or L ,  _ permet de montrer X donc on a X (sans supposer Y)
Phrases 4  : On veut montrer X ∧ Y en supposant L ,  _
 or L ,  _ permet individuellement de montrer X et aussi Y donc on a X ∧ Y
Phrases 5  : On veut montrer Y en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a Y
Phrases 6  : On veut montrer X en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a X
Phrases 7  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer Y donc on a X ∨ Y
Phrases 8  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer X donc on a X ∨ Y
Phrases 9  : On veut montrer Z en supposant L ,  _
 or L ,  _ permet de montrer X ∨ Y
et en supposant L ,  _ , X on peut montrer Z
et en supposant L ,  _ , X on peut montrer Z
 on peut donc finalement montrer Z
Phrases 10  : On veut montrer ¬X en supposant L ,  _
 or on peut montrer l'absurde (Bot) en supposant X , L ,  _ donc on a ¬X
Phrases 11  : On veut montrer que L ,  _ est absurde
```

Test de l'assistant p4

```
or on peut montrer X et ¬X en supposant L , _ donc L , _ est absurde
Phrases 12  : On veut montrer X en supposant L , _
 or on peut montrer l'absurde (Bot) en supposant L , _ et ¬X donc on a X
Phrases 13  : On veut montrer X en supposant L , _
 puis ce qu'on le suppose, on a bien X
Tapez le numéro de la phrase que vous voulez écrire :
13
Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qu
i remplace le L) :
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : A
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
1
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : B
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
0
Rentrez maintenant la formule que vous voulez pour X :
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
```

Test de l'assistant p5

```
On veut montrer A en supposant B ,  A ,  _
 puis ce qu'on le suppose, on a bien A
On veut maintenant montrer le séquent : B , A ,   ⊢B
Voici les prochaines phrases que vous pouvez utilisez :
Phrases 1  : On veut montrer X -> Y en supposant L ,  _.
 Supposons L ,  _ et X montrons Y
Phrases 2  : On veut montrer Y en supposant L ,  _.
 Or L ,  _ permet de monter X -> Y et X on a donc Y
Phrases 3  : On veut montrer X en supposant L ,  _ et Y
 or L ,  _ permet de montrer X donc on a X (sans supposer Y)
Phrases 4  : On veut montrer X ∧ Y en supposant L ,  _
 or L ,  _ permet individuellement de montrer X et aussi Y donc on a X ∧ Y
Phrases 5  : On veut montrer Y en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a Y
Phrases 6  : On veut montrer X en supposant L ,  _
 or L ,  _ permet de montrer X ∧ Y donc en particulier on a X
Phrases 7  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer Y donc on a X ∨ Y
Phrases 8  : On veut montrer X ∨ Y en supposant L ,  _
 or L ,  _ permet de montrer X donc on a X ∨ Y
Phrases 9  : On veut montrer Z en supposant L ,  _
 or L ,  _ permet de montrer X ∨ Y
et en supposant L ,  _ , X on peut montrer Z
et en supposant L ,  _ , X on peut montrer Z
 on peut donc finalement montrer Z
Phrases 10  : On veut montrer ¬X en supposant L ,  _
 or on peut montrer l'absurde (Bot) en supposant X , L ,  _ donc on a ¬X
Phrases 11  : On veut montrer que L ,  _ est absude
 or on peut montrer X et ¬X en supposant L ,  _ donc L ,  _ est absurde
Phrases 12  : On veut montrer X en supposant L ,  _
 or on peut montrer l'absurde (Bot) en supposant L ,  _ et ¬X donc on a X
Phrases 13  : On veut montrer X en supposant L ,  _
 puis ce qu'on le suppose, on a bien X
Tapez le numéro de la phrase que vous voulez écrire :
13
Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qu
i remplace le L) :
```

Test de l'assistant p6

```
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : A
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
1
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : B
Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1
0
Rentrez maintenant la formule que vous voulez pour X :
Choisissez le type de formule :
1. Variable (ex: A)
2. Non f
3. Conj(f1, f2)
4. Disj(f1, f2)
5. Impl(f1,f2)
6. Bot (toujours faux)
7. Top (toujours vrai)
1
Entrez la variable (une lettre) : B
On veut montrer B en supposant B ,  A ,  _
 puis ce qu'on le suppose, on a bien B
Votre démonstration est terminé. Demandons à notre programme si elle est correcte
Le résultat est : true
```

Code p1

```
(* paramètrage de scanf  *)



open Scanf;;


let scan_int () = Scanf.scanf " %d" (fun x -> x);;
let scan_float () = Scanf.scanf " %f" (fun x -> x);;
let scan_string () = Scanf.scanf " %s" (fun x -> x);;




(* représentation des formules pour le calcul propositionnel *)

type formule =
  |Var of char
  |Bot
  |Top
  |Impl of formule*formule
  |Neg of formule
  |Conj of formule*formule
  |Disj of formule*formule;;

type sequent =
  {
    hypotheses : formule list;
    objectif : formule;
  };;

type regle  =
  {
    premisses : sequent list;
    conclusion : sequent;
  };;

type 'a arbredemo =
  |N of regle * 'a arbredemo list;;

type demonstration = regle arbredemo;;
```

Code p2

```
(* IMPLEMENTATION DES REGLES DE DEDUCTION NATURELLE POUR LE CALCUL DES PREDICATS   *)

let intro_impl (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = f1::gamma ; objectif = f2 }] ; conclusion = {hypotheses = gamma ; objectif = Impl(f1,
f2) }}
;;


let elim_impl (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = Impl(f1,f2)} ; {hypotheses = gamma ; objectif = f1 } ] ; conclusio
n = {hypotheses = gamma  ; objectif = f2} }
;;


let affaiblissement (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = f1}] ; conclusion = {hypotheses = f2::gamma ; objectif = f1}}
;;

let intro_conj (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = f1};{hypotheses = gamma ; objectif = f2}] ; conclusion = {hypothes
es = gamma ; objectif = Conj(f1,f2)} }
;;

let elim_conj_droite (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = Conj(f1,f2)}] ; conclusion = {hypotheses = gamma ; objectif = f2}}
;;

let elim_conj_gauche (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = Conj(f1,f2)}] ; conclusion = {hypotheses = gamma ; objectif = f1}}
;;

let intro_disj_droite (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = f2}] ; conclusion = {hypotheses = gamma; objectif = Disj(f1,f2)}}
;;

let intro_disj_gauche (gamma : formule list) (f1 : formule) (f2 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = f1}] ; conclusion = {hypotheses = gamma; objectif = Disj(f1,f2)}}
;;
```

Code p3

```
let elim_disj (gamma : formule list) (f1 : formule) (f2 : formule) (f3 : formule) : regle =
  {premisses = [{hypotheses = gamma ; objectif = Disj(f1,f2)} ; {hypotheses = f1::gamma ; objectif = f3} ; {hypotheses = f2::gamma ; objectif = f3}] ; conclusion = {hypotheses = gamma; objectif = f3}}
;;

let intro_nega (gamma : formule list) (f1 : formule)  : regle =
  {premisses = [{hypotheses = f1::gamma ; objectif = Bot }] ; conclusion = {hypotheses = gamma ; objectif = Neg(f1) }}
;;

let elim_nega (gamma : formule list) (f1 : formule)  : regle =
  {premisses = [{hypotheses = gamma ; objectif = Neg(f1)} ;{hypotheses = gamma ; objectif = f1} ] ; conclusion = {hypotheses = gamma ; objectif = Bot}}
;;

let absurdite (gamma : formule list) (f1 : formule)  : regle =
  {premisses = [{hypotheses = Neg(f1)::gamma; objectif = Bot}] ; conclusion = {hypotheses = gamma; objectif = f1}}
;;

let axiome (gamma : formule list) (f1 : formule) : regle =
  {premisses = [] ; conclusion = {hypotheses = gamma ; objectif  = f1}}
;;
```

Code p4

```
(*OBJECTIF 1 : VERIFICATEUR DE PREUVE *)


(*fonctions auxiliaires  *)


let rec liste_conclu_fils l = match l with
  |[] -> []
  |a::q -> let N(r,_) = a in r.conclusion::(liste_conclu_fils q)
;;


(* ne prends pas en compte les répétitions  *)
let meme_element l1 l2 =
  let rec aux u v = match u with
    |[] -> true
    |t::q -> (List.mem t v) && (aux q v)
  in
  (aux l1 l2) && (aux l2 l1)
;;



let egalite_sequent s1 s2 =
  (s1.objectif = s2.objectif) && (meme_element s2.hypotheses s1.hypotheses)
;;


(* prends en compte les répétitions  *)
let rec meme_element2 l1 l2 = match l1,l2 with
  |[],[] -> true
  |[e1],[e2] -> egalite_sequent e1 e2
  |_,[] -> false
  |[],_ -> false
  |t1::q1,t2::q2 -> if (egalite_sequent t1 t2) then meme_element2 q1 q2
                    else meme_element2 (q1@[t1]) l2
;;
```

Code p5

```
let jonction_correcte regle liste_fils =
  meme_element2 (liste_conclu_fils (liste_fils)) regle.premisses;;


let rec list_all_true f l = match l with
  |[] -> true
  |t::q -> (f t) && (list_all_true f q)
;;


(* finalement notre fonction de vérification  *)


let rec est_valide (d : demonstration) : bool = match d with
  |N(r,[])->  (List.mem r.conclusion.objectif r.conclusion.hypotheses)
  |N(r,l) -> (jonction_correcte r l) && (list_all_true est_valide l)
;;
```

Code p6

```
(* Code pour nombreux test construction est_valide  *)



(*Première exemple pour une preuve simple et courte : non(A) -> (A -> bot)   *)

(* modèle N( regle , [liste de regle suivante] ) avec regle : premisses -> sequent list et conclusion -> sequent av
ec sequent : hypotheses -> formule list et objectif -> formule *)


let (demo1 : demonstration) = N(  (intro_impl [] (Neg(Var('A'))) (Impl(Var('A'),Bot)) )

                                , [ N( (intro_impl [Neg(Var('A'))] (Var('A')) (Bot))

                                , [ N( (elim_nega [Neg(Var('A')) ; Var('A')] (Var('A'))  )

                                ,  [ N({premisses = [] ; conclusion = {hypotheses = [Var('A') ; Neg(Var('A')) ] ; o
bjectif = (Var('A')) } }  ,[])
                                ;
                                  N( {premisses = [] ; conclusion = {hypotheses = [Var('A') ; Neg(Var('A')) ] ; obj
ectif = (Neg(Var('A'))) } }  ,[])    ]
                                )
                                ]
                                )
                                ]
                                );;


let r1 = intro_impl [] (Neg(Var('A'))) (Impl(Var('A'),Bot));;


let r2 = intro_impl [Neg(Var('A'))] (Var('A')) (Bot);;


let r3 = elim_nega [Neg(Var('A')) ; Var('A')] (Var('A'));;


let r4 = {premisses = [] ; conclusion = {hypotheses = [Var('A') ; Neg(Var('A')) ] ; objectif = (Var('A')) } };;


let r5 = {premisses = [] ; conclusion = {hypotheses = [Var('A') ; Neg(Var('A')) ] ; objectif = (Neg(Var('A'))) } };
;
```

Code p7

```
let demo1autredef = N(r1,[N(r2,[N(r3,[N(r4,[]);N(r5,[])])])]);;

let ss_arbre1 = N(r2,[N(r3,[N(r4,[]);N(r5,[])])]);;

let ss_arbre2 = N(r3,[N(r4,[]) ; N(r5,[]) ]);;

let ss_arbre3 = N(r4,[]);;

let ss_arbre4 = N(r5,[]);;


let (demorien : demonstration) = N({premisses = [] ; conclusion = {hypotheses = [Var('A')] ; objectif = Var('A')}},
[]);;


let (demotest : demonstration) = N(
                                   {
                                      premisses = [ {hypotheses = [Var('A')] ; objectif = Var('A')}  ; {hypotheses
= [Var('B')] ; objectif = Var('B')}                      ]

                                     ;

                                      conclusion = {  hypotheses = [] ; objectif =  (Conj((Var('A')),(Var('B'))))
}



                                   },



                                   [

                                    N({premisses = [] ; conclusion = {hypotheses = [Var('A')] ; objectif = (Var('A
'))}},[])
                                   ;
                                    N({premisses = [] ; conclusion = {hypotheses = [Var('B')] ; objectif = (Var('B
'))}},[])
                                   ]
                                   )
;;
```

Code p8

```
let r11 = intro_conj [Var('A') ; Var('B')] (Var('A')) (Var('B'));;

let r12 = {premisses = [] ; conclusion = {hypotheses =[Var('A') ; Var('B')] ; objectif = (Var('A'))}};;

let r13 = {premisses = [] ; conclusion = {hypotheses =[Var('A') ; Var('B')] ; objectif = (Var('B'))}};;



let demo_presentation = N(r11,[N(r12,[]) ; N(r13,[])]);;




(* FIN OBJECTIF 1   *)
```

Code p9

```
(* OBJECTIF 2 : Assistant/traduction de preuve  *)



(* fonctions auxiliaire pour mise en place de la traduction des regles *)

let rec form_to_str (f : formule) : string = match f with
  |Top -> "Top"
  |Bot -> "Bot"
  |Var(c) -> String.make 1 c
  |Conj(f1,f2) ->  (form_to_str f1) ^ " et " ^ (form_to_str f2)
  |Disj(f1,f2) ->  (form_to_str f1) ^ " ou " ^ (form_to_str f2)
  |Impl(f1,f2) -> (form_to_str f1) ^ " -> " ^ (form_to_str f2)
  |Neg(f1) -> "Non(" ^ (form_to_str f1) ^ ")"
;;

let rec list_form_to_str l = match l with
  |[] -> "_"
  |t::q -> (form_to_str t) ^ " ,  " ^ (list_form_to_str q)
;;


(*Traduction des preuves  *)


let intro_impl_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s -> %s en supposant %s. \n Supposons %s et %s montrons %s  " s1 s2 s3 s3 s1 s2;
;
```

Code p10

```
let elim_impl_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s. \n Or %s permet de monter %s -> %s et %s on a donc %s " s2 s3
 s3 s1 s2 s1 s2 ;;


let affaiblissement_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s et %s \n or %s permet de montrer %s donc on a %s (sans suppose
r %s)" s1 s3 s2 s3 s1 s1 s2;;


let intro_conj_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s ∧ %s en supposant %s \n or %s permet individuellement de montrer %s et aussi %
s donc on a %s ∧ %s" s1 s2 s3 s3 s1 s2 s1 s2;;
;;


let elim_conj_droite_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n or %s permet de montrer %s ∧ %s donc en particulier on a %s
" s2 s3 s3 s1 s2 s2;;
```

Code p11

```
let elim_conj_gauche_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n or %s permet de montrer %s ∧ %s donc en particulier on a %s
" s1 s3 s3 s1 s2 s1;;


let intro_disj_droite_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s ∨ %s en supposant %s \n or %s permet de montrer %s donc on a %s ∨ %s" s1 s2 s3
 s3 s2 s1 s2;;


let intro_disj_gauche_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s ∨ %s en supposant %s \n or %s permet de montrer %s donc on a %s ∨ %s" s1 s2 s3
 s3 s1 s1 s2;;


let elim_disj_fr (gamma : formule list) (f1 : formule) (f2 : formule) (f3 : formule) : string  =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (form_to_str f3) in
  let s4 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n or %s permet de montrer %s ∨ %s \net en supposant %s , %s o
n peut montrer %s \net en supposant %s , %s on peut montrer %s \n on peut donc finalement montrer %s " s3 s4 s4 s1
s2 s4 s1 s3 s4 s1 s3 s3;;
```

Code p12

```
let intro_disj_gauche_fr (gamma : formule list) (f1 : formule) (f2 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s ∨ %s en supposant %s \n or %s permet de montrer %s donc on a %s ∨ %s" s1 s2 s3 s3 s1 s1 s2;;


let elim_disj_fr (gamma : formule list) (f1 : formule) (f2 : formule) (f3 : formule) : string  =
  let s1 = (form_to_str f1) in
  let s2 = (form_to_str f2) in
  let s3 = (form_to_str f3) in
  let s4 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n or %s permet de montrer %s ∨ %s \net en supposant %s , %s on peut montrer %s \net en supposant %s , %s on peut montrer %s \n on peut donc finalement montrer %s " s3 s4 s4 s1 s2 s4 s1 s3 s4 s1 s3 s3;;


let intro_nega_fr (gamma : formule list) (f1 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer ¬%s en supposant %s \n or on peut montrer l'absurde (Bot) en supposant %s , %s donc on a ¬%s" s1 s2 s1 s2 s1;;


let elim_nega_fr (gamma : formule list) (f1 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer que %s est absude \n or on peut montrer %s et ¬%s en supposant %s donc %s est absurde" s2 s1 s1 s2 s2;;
```

Code p13

```
let absurdite_fr (gamma : formule list) (f1 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n or on peut montrer l'absurde (Bot) en supposant %s et ¬%s donc on a %s" s1 s2 s2 s1 s1;;


let axiome_fr (gamma : formule list) (f1 : formule) : string =
  let s1 = (form_to_str f1) in
  let s2 = (list_form_to_str gamma) in
  Printf.sprintf "On veut montrer %s en supposant %s \n puis ce qu'on le suppose, on a bien %s " s1 s2 s1;;


(*TABLEAU FONCTIONS POUR Y ACCEDER PLUS FACILEMENT  *)

let tab_regle0_fr = [| intro_impl_fr ; elim_impl_fr ; affaiblissement_fr ; intro_conj_fr ; elim_conj_droite_fr ; elim_conj_gauche_fr ;  intro_disj_droite_fr ;  intro_disj_gauche_fr |];;

let tab_regle1_fr = [| elim_disj_fr |];;

let tab_regle2_fr = [| intro_nega_fr  ; elim_nega_fr ; absurdite_fr  ; axiome_fr |];;



let tab_regle0 = [| intro_impl ; elim_impl ; affaiblissement ; intro_conj ; elim_conj_droite ; elim_conj_gauche ;  intro_disj_droite ;  intro_disj_gauche  |];;

let tab_regle1 = [|elim_disj|];;

let tab_regle2 = [| intro_nega  ; elim_nega ; absurdite ; axiome|];;
```

Code p14

```
let rec construct_formule () : formule =
  print_endline "Choisissez le type de formule :";
  print_endline "1. Variable (ex: A)";
  print_endline "2. Non f";
  print_endline "3. Conj(f1, f2)";
  print_endline "4. Disj(f1, f2)";
  print_endline "5. Impl(f1,f2)";
  print_endline "6. Bot (toujours faux)";
  print_endline "7. Top (toujours vrai)";
  flush stdout;
  match scan_int() with
  | 1 ->
      print_string "Entrez la variable (une lettre) : ";
      flush stdout;
      let c = scan_string () in
      Var(c.[0])
  | 2 ->
      print_endline "Construisons la formule f :";
      let f = construct_formule () in
      Neg(f)
  | 3 ->
      print_endline "Construisons la formule f1 :";
      let f1 = construct_formule () in
      print_endline "Construisons la formule f2 :";
      let f2 = construct_formule () in
      Conj(f1, f2)
  | 4 ->
      print_endline "Construisons la formule f1 :";
      let f1 = construct_formule () in
      print_endline "Construisons la formule f2 :";
      let f2 = construct_formule () in
      Disj(f1, f2)
  | 5 ->
      print_endline "Construisons la formule f1 :";
      let f1 = construct_formule () in
      print_endline "Construisons la formule f2 :";
      let f2 = construct_formule () in
      Impl(f1, f2)
  | 6 -> Bot
  | 7 -> Top
  | _ ->
      print_endline "Choix invalide.";
      construct_formule ()
;;
```

Code p15

```
(*Construction de notre fonction récursive qui va permettre à l'utilisateur de construire sa démonstration  *)

let rec list_formule_to_str l = match l with
  |[] -> " "
  |t::q -> (form_to_str t) ^ " , " ^ (list_formule_to_str q)
;;

let print_sequent (s : sequent) : unit =
  print_string (list_formule_to_str s.hypotheses);
  print_string " ⊢ ";
  print_string (form_to_str (s.objectif));
;;


(*regle par défault pour faciliter implémentation  *)

let regle_null = {premisses = [] ; conclusion = {hypotheses = []; objectif = Bot}};;




(*Il faut prouver toutes les premisses de la regle   *)
```

Code p16

```
let rec construction_demo (s : sequent) : demonstration =

  print_string "On veut maintenant montrer le séquent : ";
  print_sequent s;
  print_newline();


  print_endline "Voici les prochaines phrases que vous pouvez utilisez : ";
  for i=0 to 7 do
    Printf.printf  "Phrases %d  : " (i+1);
    print_endline (tab_regle0_fr.(i) [Var('L')]  (Var('X')) (Var('Y')) );
  done;
  Printf.printf  "Phrases %d  : " 9;
  print_endline (tab_regle1_fr.(0)  [Var('L')]   (Var('X'))  (Var('Y'))  (Var('Z')) );
  for i = 0 to 3 do
    Printf.printf  "Phrases %d  : " (i+10);
    print_endline (tab_regle2_fr.(i) [Var('L')] (Var('X')) );
  done;

  Printf.printf "Tapez le numéro de la phrase que vous voulez écrire :\n";
  flush stdout;
  let choix = scan_int() in
```

Code p17

```
if (choix<9) then

  begin

  print_endline "Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qui rempl
ace le L) : ";
  flush stdout;
  let l = ref [] in
  let fin_liste = ref 1 in
  print_endline "Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1";
  flush stdout;
  fin_liste := scan_int();
  while (!fin_liste = 1) do
    let f = construct_formule() in
    l := f::(!l);
    print_endline "Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1";
    flush stdout;
    fin_liste := scan_int();
  done;
  print_endline "Rentrez maintenant la formule que vous voulez pour X : ";
  flush stdout;
  let f1 = construct_formule() in
  print_endline "Rentrez maintenant la formule que vous voulez pour Y : ";
  flush stdout;
  let f2 = construct_formule() in

  (* Afficher le résultat de la règle utiliser avec les entrées de l'utilisateur *)

  print_endline (tab_regle0_fr.(choix-1) (!l) f1 f2);
  flush stdout;

    (* Ajouter à notre arbre de preuve la règle avec les bonnes entrées  *)
  let r = (tab_regle0.(choix-1)  (!l) (f1) (f2)) in

  N( r , List.map  construction_demo (r.premisses)  )


  end
```

Code p18

```
else if choix = 9 then
  begin

  print_endline "Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qui remplace le L) : ";
  flush stdout;
  let l = ref [] in
  let fin_liste = ref 1 in
  print_endline "Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1";
  fin_liste := scan_int();
  while (!fin_liste = 1) do
    let f = construct_formule() in
    l := f::(!l);
    print_endline "Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1";
    flush stdout;
    fin_liste := scan_int();
  done;
  print_endline "Rentrez maintenant la formule que vous voulez pour X : ";
  flush stdout;
  let f1 = construct_formule() in
  print_endline "Rentrez maintenant la formule que vous voulez pour Y : ";
  flush stdout;
  let f2 = construct_formule() in
  print_endline "Rentrez maintenant la formule que vous voulez pour Z : ";
  flush stdout;
  let f3 = construct_formule() in

  print_endline (tab_regle1_fr.(0) (!l) f1 f2 f3);
  flush stdout;

  let r = (tab_regle1.(0) (!l) f1 f2 f3) in

  N(r, List.map construction_demo (r.premisses))

  end
```

Code p19

```
else if choix>9 && choix<14 then

  begin

  print_endline "Rentrez premièrement la liste des formules que vous voulez prendre pour hypothèses (ce qui remplace le L) : ";
  flush stdout;
  let l = ref [] in
  let fin_liste = ref 1 in
  while (!fin_liste = 1) do
    let f = construct_formule() in
    l := f::(!l);
    print_endline "Si vous avez fini de rentrer vos hypothèses Tapez 0 sinon Tapez 1";
    flush stdout;
    fin_liste := scan_int();
  done;
  print_endline "Rentrez maintenant la formule que vous voulez pour X : ";
  flush stdout;
  let f1 = construct_formule() in

  print_endline (tab_regle2_fr.(choix-10) (!l) f1);
  flush stdout;

  let r = (tab_regle2.(choix-10) (!l) f1) in

  N(r, List.map construction_demo (r.premisses))

  end

else

  begin

  print_endline "Erreur : choix d'une phrase non existente, recommencez tout !";
  N(regle_null,[])

  end
```

Code p20

```
let assistant (s : sequent) : bool =
  print_string "L'assistant va vous aider à montrer le séquent : ";
  print_sequent s;
  print_newline();
  print_endline "Puis va vous indiquez si votre preuve est correcte";
  let d = construction_demo s in
  print_endline "Votre démonstration est terminé. Demandons à notre programme si elle est correcte";
  let b = est_valide d in
  Printf.printf "Le résultat est : %b\n" b;
  b
;;


assistant {hypotheses = [Var('A');Var('B')] ; objectif = Conj(Var('A'),Var('B'))};;




(*FIN OBJECTIF 2  *)
```