NFP108-NFP120 Logique et sémantique des programmes

19 février 2015

Table des matières

Chapitre 1

introduction

1.1 Objectif du document

Ce document sert de support aux cours NFP108 (partie logique) et NFP120 (parties logique et sémantique des programmes) du CNAM. L'objectif est double :

- faire une introduction à la logique, en particulier les notions de syntaxe, de sémantique et de système de déduction ;
- mettre en évidence les liens entre la logique et l'étude des langages de programmation en abordant les mêmes notions de syntaxe, sémantique et système d'inférence pour ces langages. En effet un programme peut être vu comme une *formule* avec une syntaxe et une sémantique particulières.

Le parti pris de ce document est de définir toutes les notions logiques dans l'assistant de preuve Coq [1]. Ces définitions sont donc écrites dans un langage formel dont vous n'avez probablement pas l'habitude. Le but de cette introduction est de fournir une bibliographie ainsi qu'un lexique permettant d'appréhender ce langage formel.

IL NE S'AGIT PAS d'un cours d'utilisation de Coq (ce n'est pas au programme), uniquement d'une aide à la lecture du document.

1.2 Que puis-je lire concernant ce cours ?

Les supports de cours officiels sont sur le site du cours. Il s'agit souvent simplement de la version PDF des fichiers Coq présentés en cours. Le présent lexique est une aide à la lecture de ces fichiers. Ci-dessous une liste de références couvrant le contenu du cours et permettant d'approfondir les sujet abordés.

1.2.1 Concernant la logique propositionnelle et la logique des prédicats

La logique des prédicats intègre en principe également les symboles de fonctions, ce que nous ne faisons pas dans ce cours. Vous pouvez donc ignorer cet aspect : les termes ne sont que des variables.

- fr.wikipedia.org/wiki/Calcul_des_propositions en particulier la section « Sémantique » (vous pouvez ignorer la partie « systèmes déductifs »).

1. http://coq.inria.fr/

— fr.wikipedia.org/wiki/Calcul_des_prédicats
— Sur la notion d'interprétation des formules : fr.wikipedia.org/wiki/Théorie_des_modèles
— Vous pouvez regarder également n'importe quel livre de logique, en général les premiers chapitre concerne la logique propositionnelle et les prédicats.
— R. Lassaigne et M. de Rougemont. « Logique et fondements de l'informatique ». Hermes, 1993.
— R. Cori et D. Lascar. « Logique mathématique : Calcul propositionnel, algèbres de Boole, calcul des prédicats ». Masson 1993.
— R. David, K. Nour et C. Raffalli. « Introduction à la logique, Théorie de la démonstration ». Dunod 2004.
— P. Lafourcade, M. Lévy et S. Desvismes. « Logique et démonstration automatique, Informatique théorique - Introduction à la logique propositionnelle et à la logique du premier ordre (Niveau A) ». Ellipses 2012.

1.2.2 Concernant la méthode des tableaux

— fr.wikipedia.org/wiki/Méthode_des_tableaux

Sans regarder la section « metavariables et unification » qui est hors-sujet pour nous.

1.2.3 Concernant la sémantique des programmes

— Robert Harper. Practical Foundations for Programming Languages. MIT Press.
— J. C. Mitchell. Foundations for Programming Languages. MIT Press, 1996.
— B. C. Pierce. Types and Programming Languages. MIT Press, 2002.
— B. C. Pierce et al,
Software Foundations : www.cis.upenn.edu/~bcpierce/sf/index.html. En Coq.

1.2.4 Concernant la preuve de programme

— G. Winskel. « The Formal Semantics of Programming Languages : An Introduction ». MIT Press, 1993.
— Les documentations des outils de preuve de programmes : Frama-C, Spark/ADA, Microsoft Boogie, Java Extended Static Checking, etc.

1.3 Qu'est-ce-que Coq ?

Coq [2] est un *assistant de preuve*. Autrement dit c'est un logiciel dans lequel l'utilisateur peut écrire des définitions mathématiques formelles, dans un style ressemblant à la programmation, et effectuer des démonstrations sur ces définitions. Le cadre formel est contraignant et la syntaxe rigide mais assurent que les démonstrations sont correctes.

1.4 Pourquoi utiliser Coq dans ce cours ?

Parce-que l'expérience montre que les mathématiques formelles sont plus compréhensibles (plus palpables), en particulier pour un public d'informaticiens, quand elles sont présentées sous cette forme proche de la programmation. Par ailleurs c'est un moyen pour l'enseignant de s'assurer qu'il ne dit que des choses correctes d'un point de vue logique...

2. coq.inria.fr

Dans sa version précédente ce cours utilisait Prolog pour la même raison. Nous avons décidé de passer à Coq pour deux raisons : d'une part le langage Prolog est passé de mode (ce qui n'implique pas un jugement de valeur de la part de l'auteur) et d'autre part parce-que la sémantique de Prolog est difficile à appréhender et prend trop de temps. NFP120 initialement était un cours d'un an et pas un semestre, ce qui laissait du temps pour comprendre Prolog avant de s'en servir pour définir et programmer les autres aspects du cours.

Dans ce cours il ne vous sera pas demandé de savoir utiliser Coq, seulement de savoir lire les définitions contenues dans un fichier Coq. Le but de ce lexique est de vous y aider.

Chapitre 2

lexique

2.1 forall, exists, /\, \/, $\longrightarrow$, etc

Dans ce cours on procédera de la manière suivante pour étudier les différentes logiques au programme :

- premièrement définition syntaxique de la notion de formule sous la forme d'un type `formule` défini en Coq ;
- définition de la notion de sémantique d'une formule ;
- définitions et démonstrations des propriétés de la sémantique.

Il s'agit donc de définir mathématiquement la notion de formule. Pour cela nous avons besoin du langage mathématique qui utilise lui aussi sa propre notion de formule.

Attention il y a donc deux types de formules : d'une part les formules Coq qui permettent de définir les propriétés dans le langage formel de Coq, et d'autre part les éléments du type `formule` que nous définissons dans le langage de Coq et sur lequel nos définitions porterons. Dans la mesure du possible nous essayons d'avoir deux jeux de connecteurs distincts.

Dans ce cours les symboles logiques en caractères ASCII seront utilisés pour les formules Coq (`forall` pour la quantification universelle, ~ la négation, /\ pour la conjonction (et), \/ pour la disjonction (ou) etc) à l'exception de l'implication $\longrightarrow$. Nous réserverons les notations symboliques non ASCII ($\forall$, $\exists$, $\wedge$, $\vee$, etc l'implication étant différenciée par la double flèche $\Rightarrow$) pour les éléments du type `formule` (voir plus bas section Inductive, en particulier le deuxième exemple).

Il est à noter que dans leur versions html, pdf ou texte les symboles n'ont pas exactement le même aspect.

2.2 Definition f :T := def.

Définit la constante `f`, de type `T` dont la définition est `def`. Il s'agit d'une syntaxe semblable à celle des langages de programmation. Lorsque `f` est une fonction, les arguments de `f` sont mis en paramètres de la manière suivante : `Definition f (p1 :t1) ... (pn :t1) : T := def.`, où les `pi` sont les noms des paramètres et les `ti` désignent leurs types.

Par exemple ci dessous la définition d'une fonction `table_Non` prend un booléen en argument et retourne le booléen opposé. Notez au passage la construction `match ... with ...` comparable dans une certaine mesure à la commande `switch` de Java/C.

```
Definition table_Non(x :bool) :bool :=
  match x with
    | true => false
    | false => true
  end.
```

Pour définir une fonction récursive, on utilisera la construction `Fixpoint` (ou de manière équivalente `Function`).

2.3 `Fixpoint f (x1 :t1) ... (xn :tn) : T := def.` ou `Function f (x1 :t1) ... (xn :tn) : T := def.`

Définition d'une fonction récursive. La syntaxe est comparable au `Definition f (p1 :t1) ... (pn :t1) : T := def.` ci-dessus, excepté que `f` peut apparaître dnas sa propre définition (appel récursif).

2.4 `Lemma X : Y.` ou `Theorem X :Y.`

Démarrage de la démonstration de la propriété `Y`. Le nom du théorème (du lemme) une fois prouvé sera `X`. La preuve est ensuite suivie d'un ensemble de \emph{tactiques} `Proof. ... End.` que vous n'avez pas à comprendre (hors sujet pour NFP120, généralement masquées dans les documents pdf et html).

2.5 `Notation "xxx" := (yyy)`

Définit une notation `xxx` pour écrire `yyy` d'une manière plus agréable. Ceci est très important, nous utiliserons dans la mesure du possible les notations mathématiques usuelles pour la logique et la sémantique. Souvent après la définition formelle d'une notion nous lui adjoindrons une notation et nous utiliserons celle-ci partout dans la suite. Voir par exemple plus bas dans la section « Deuxième exemple ».

2.6 Commandes usuelles

2.6.1 `Check x` ou `Print x.`

Demande au système le type ou la valeur de `x`. En général la réponse obtenue est ajouté dans le document juste après la commande.

2.6.2 `Module X.`, `End X.`, `Import X.` ou `Require X.`

Commandes de structuration des fichiers, vous pouvez les ignorer.

2.6.3 `Add Morphism ....` ou `Add Relation ...`

Commandes permettant de faciliter les preuves, vous pouvez les ignorer.

2.7 `Inductive I : T := def`

Définit le type (l'ensemble) `I` par induction à l'aide des opérateurs donnés dans `def`. Nous expliquons rapidement plus bas ce que signifie une définition par induction.

2.7.1 Définition d'ensemble par induction

Il existe 3 méthodes canoniques pour définir un ensemble E, toutes possibles dans coq :
- *Par extension* : On donne la liste exhaustive (*extensive*) des éléments. Par exemple : $E = \{0, 1, 2, 3\}$.
- *Par intention* : On donne la propriété qui caractérise les éléments de l'ensemble. Par exemple : $E = \{x | x \in \mathbb{N} \wedge x \leq 3\}$.
- *Par induction* : nous détaillons cette méthode ci-dessous. Dans ce cas on décrit l'ensemble des opérations permettant de *construire* (d'énumérer, même indéfiniment) tous les éléments de l'ensemble.

2.7.2 Premier exemple : définition de $\mathbb{N}$ par induction

Comment définir l'ensemble $\mathbb{N}$ des nombres entiers naturels ? Par extension c'est impossible puisque l'ensemble est infini. Par intention c'est possible si on a déjà défini un sur-ensemble ($\mathbb{Z}$ ou $\mathbb{R}$ par exemple) mais sinon c'est également impossible.

Intuitivement on écrirait $\mathbb{N}$=$\{0, 1, 2, ...\}$. Il ne s'agit évidemment pas d'une définition correcte puisqu'elle n'exhibe que trois entiers. Les points de suspension ne font pas partie du langage mathématique formel.

On peut néanmoins définir $\mathbb{N}$ de façon rigoureuse en suivant cette intuition, en définissant non pas directement les éléments de $\mathbb{N}$ mais plutôt les *opérateurs* permettant de construire tous ses éléments.

On définit l'ensemble $\mathbb{N}$ par induction de la façon suivante :
- l'élément 0 appartient à $\mathbb{N}$ (0∈$\mathbb{N}$)
- Si n∈$\mathbb{N}$ alors succ(n)∈$\mathbb{N}$
- $\mathbb{N}$ est le plus petit ensemble clos par 0 et succ.

Autrement dit $\mathbb{N}$ contient les éléments suivants :

0, succ(0), succ(succ(0)), succ(succ(succ(0))), ...

Attention l'opérateur succ peut être vu soit comme une fonction (telle que succ(n) retourne la valeur de n+1) soit comme un simple constructeur c'est-à-dire que succ(n) est lui-même un élément de l'ensemble défini et ne se calcule pas. Dans la syntaxe du logiciel Coq on écrira :

```
Inductive nat : Type :=
    O : nat
| succ : nat ⟶ nat.
```

`Inductive nat :Type` signifie qu'on défini un nouvel ensemble (un type) $\mathbb{N}$ par induction. `O :nat` signifie que `O` est un constructeur à zéro argument, et `succ : nat ⟶ nat` signifie que `succ` est un constructeur à un argument de type $\mathbb{N}$.

2.7.3 Deuxième exemple : les formules propositionnelles

De la même manière que $\mathbb{N}$ peut être défini par induction, l'ensemble des formules propositionnelles (sans variable), noté F_p peut l'être aussi :
- $\top \in F_p$

— $\bot \in F_p$
— si $f \in F_p$ alors $\neg f \in F_p$
— si $f_1, f_2 \in F_p$ alors $f_1 \vee f_2 \in F_p$
— si $f_1, f_2 \in F_p$ alors $f_1 \wedge f_2 \in F_p$
— si $f_1, f_2 \in F_p$ alors $f_1 \Rightarrow f_2 \in F_p$
— F_p est le plus petit ensemble clos par les opérateurs $\top$, $\bot$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$.

Notez qu'on peut parler de *grammaire* des formules, au sens où cette définition inductive définit les règles de bonne formation des formules. On voit donc souvent la définition ci-dessus exprimée de la façon suivante (dite grammaire BNF [1]) :

$F_p ::= \top \mid \bot \mid \neg F_p \mid F_p \vee F_p \mid F_p \wedge F_p \mid F_p \Rightarrow F_p$

Dans la syntaxe Coq on définit F_p comme un type inductif `formule` comme suit (voir également les fichiers Coq) :

```
Inductive formule : Type :=
  | Vrai : formule
  | Faux : formule
  | Non : formule → formule
  | Ou : formule → formule → formule
  | Et : formule → formule → formule
  | Implique : formule → formule → formule.
```

Des notations définies a posteriori permettent d'utiliser les symboles usuels ($\top$ pour `Vrai`, $\bot$ pour `Faux`, $\neg$`f` pour `Non(f)`, `f`$\vee$`g` pour `Ou(f,g)` etc).

```
Notation "⊤" := Vrai.
Notation "⊥" := Faux.
Notation "¬ X" := (Non X).
Notation "X ∨ Y" := (Ou X Y).
Notation "X ∧ Y" := (Et X Y).
Notation "X ⇒ Y" := (Implique X Y).
```

2.7.4 Troisième exemple : la propriété inductive "interprétation d'une formule"

En plus des ensembles, les définitions inductives permettent également de définir des propriétés ou des relations. Par exemple nous donnons ici la définition de la relation `I f b` qui est vrai lorsque le booléen `b` est l'interprétation de la formule propositionnelle `f` (Où !, && et || sont les opérateurs booléens usuels).

— `I` $\top$ `true`
— `I` $\bot$ `false`
— Si `I f b` alors `I` ($\neg$ `f`) (`!b`)
— Si `I` f_1 b_1 et `I` f_2 b_1 alors `I` ($f_1 \vee f_2$) (`b1 || b2`)
— Si `I` f_1 b_1 et `I` f_2 b_2 alors `I` ($f_1 \wedge f_2$) (f_1 `&&` f_2)
— Si `I` f_1 b_1 et `I` f_2 b_2 alors `I` ($f_1 \Rightarrow f_2$) (f_2 `||!` f_1)

En syntaxe Coq cela donne :

```
Inductive I : formule → bool → Prop :=
| I_Vrai : I ⊤ true
| I_Faux : I ⊥ false
| I_Non : forall f b1 b, I f b1 → !b1 = b → I (¬f) b
```

1. http://fr.wikipedia.org/wiki/Forme_de_Backus-Naur

```
| I_Ou : forall f1 f2 b1 b2 b, I f1 b1 ⟶ I f2 b2 ⟶ b1 || b2 = b ⟶ I (f1 ∨ f2) b
| I_Et : forall f1 f2 b1 b2 b, I f1 b1 ⟶ I f2 b2 ⟶ b1 && b2 = b ⟶ I (f1 ∧ f2) b
| I_Implique : forall f1 f2 b1 b2 b, I f1 b1 ⟶ I f2 b2 ⟶ (!b1) || b2 = b ⟶ (I (f1 ⇒ f2) b).
```

Il y a un exemple (plus compliqué) de définition semblable dans le développement Coq sur la logique des prédicats avec quantificateurs. Il y a aussi un exemple similaire dans la partie sur la sémantique des programmes.

Il faut lire une telle définition de la manière suivante : pour que la propriété `I f b` soit vraie, il faut qu'il existe une combinaison des opérateurs ayant comme conclusion `I f b`.

Cette notion de combinaison d'opérateur (appelée également *arbre d'inférence*, *arbre de dérivation*, *arbre de preuve* etc), fera l'objet de plusieurs séances de cours.

Chapitre 3

tables_de_verite

3.1 Les tables de vérité des connecteurs

Les tables de vérité des opérateurs booléens sont des fonctions des booléens vers les booléens (le nombre d'arguments étant égal à l'arité du connecteur). Elle décrivent le résultat de l'application du connecteur sur des *booléens*. On généralisera le calcul d'un booléen à partir d'une *formule* à l'aide de la notion d'interprétation. On présente une table de vérité en général par une grille énumérant toutes les valeurs possibles pour les arguments (une valuation des argument par ligne) et pour chaque ligne la réponse du connecteur.

true et false sont des connecteurs à 0 arguments.

```
 V
---
 V
```

```
Definition table_Vrai :bool := true.
```

```
 F
---
 F
```

```
Definition table_Faux :bool := false.
```

La négation est un connecteurs à 1 argument.

```
 x  ¬(x)
--------
 V   F
 F   V
```

```
Function table_Non(x :bool) :bool :=
  match x with
    | true ⇒ false
    | false ⇒ true
  end.
```

Les autres connecteurs sont binaires.

```
 x  y  x∨y
----------
 V  V   V
 V  F   V
 F  V   V
 F  F   F
```

```
Function table_Ou(x y :bool) :bool :=
  match x,y with
    | true,true ⇒ true
    | true,false ⇒ true
    | false,true ⇒ true
    | false,false ⇒ false
  end.
```

```
 x  y  x∧y
----------
 V  V   V
 V  F   F
 F  V   F
 F  F   F
```

```
Function table_Et(x y :bool) :bool :=
  match x,y with
    | true,true ⇒ true
    | true,false ⇒ false
    | false,true ⇒ false
    | false,false ⇒ false
  end.
```

La table de vérité de l'implication est parfois source d'erreur chez le débutant. Remarquez que lorsque x est faux, x → y est vraie indépendamment de y.

```
 x  y  x->y
----------
 V  V   V
 V  F   F
 F  V   V
 F  F   V
```

```
Function table_Implique(x y :bool) :bool :=
  match x,y with
    | true,true ⇒ true
    | true,false ⇒ false
    | false,_ ⇒ true
  end.
```

Chapitre 4

logique_propositionnelle

Ce module formalise la logique propositionnelle sans variable (calcul propositionnel).

4.1 Les formules propositionnelles (sans variables)

4.1.1 Défintion des formules

Les formules sont définies (par induction) par la grammaire suivante.

```
Inductive formule : Type :=
| Vrai : formule
| Faux : formule
| Non : formule → formule
| Ou : formule → formule → formule
| Et : formule → formule → formule
| Implique : formule → formule → formule.
```

4.1.2 Exemples de formule sans notation

```
Check Vrai.
Check Faux.
Check (Ou Vrai Vrai).
Check (Ou Faux Vrai).
Check (Ou (Ou Vrai Faux) Vrai).
```

4.1.3 Notations usuelles

```
Notation "⊤" := Vrai.
Notation "⊥" := Faux.
Notation "¬ X" := (Non X).
Notation "X ∨ Y" := (Ou X Y).
Notation "X ∧ Y" := (Et X Y).
Notation "X ⇒ Y" := (Implique X Y).
```

4.1.4 Exemples avec les notations usuelles

```
Check ⊤.
Check ⊥.
Check (⊤ ∨ ⊤).
Check (⊥ ∨ ⊤).
Check ((⊤ ∨ ⊥) ∨ ⊤).
Check (⊤ ∨ ⊥ ∨ ⊤).
Check (⊤ ∨ (⊥ ∨ ⊤)).
```

4.2 Interprétation d'une formule (Sémantique)

4.2.1 Définition de l'interprétation d'une formule

Définition de l'interprétation d'une formule : on applique les tables de vérité des feuilles jusqu'à la racine.

```
Function interp_def (f :formule) : bool :=
  match f with
    | ⊤ => table_Vrai
    | ⊥ => table_Faux
    | ¬ f => table_Non (interp_def f)
    | f1 ∨ f2 => table_Ou (interp_def f1) (interp_def f2)
    | f1 ∧ f2 => table_Et (interp_def f1) (interp_def f2)
    | f1 ⇒ f2 => table_Implique (interp_def f1) (interp_def f2)
  end.

Eval compute in interp_def (Implique ⊤ (Ou ⊤ ⊥)).
Eval compute in interp_def (Implique ⊤ (Ou ⊥ ⊥)).
```

Version "optimisée" qui n'évalue la sous-formule de droite que si nécessaire.

```
Function interp (f :formule) : bool :=
  match f with
    | ⊤ => true
    | ⊥ => false
    | ¬f => if interp f then false else true
    | f1 ∨ f2 => if interp f1 then true else interp f2
    | f1 ∧ f2 => if interp f1 then interp f2 else false
    | f1 ⇒ f2 => if interp f1 then interp f2 else true
  end.

Eval compute in interp (Implique ⊤ (Ou ⊤ ⊥)).
Eval compute in interp (Ou ⊥ ⊥).
```

Preuve de correction de la version optimisée.

```
Lemma interp_correct : forall f :formule, interp_def f = interp f.
```

4.3 Preuves d'équivalences entre formules

Dans le calcul propositionnel deux formules sont équivalentes si leurs interprétations sont égales. Dans les logiques avec variables nous verrons que la notion d'équivalence fait intervenir la notion de *valuation* des variables.

```
Notation "f ≡ g" := ((interp f) = (interp g)) (at level 180).
Lemma eq_implique : forall x y : formule, x ⇒ y ≡ ¬x ∨ y.
Lemma eq_et : forall x y : formule, x ∧ y ≡ ¬(¬x ∨ ¬y).
Lemma eq_not : ⊥ ≡ ¬⊤.
```

4.4 Réduction des connecteurs au sous-ensemble {⊤,¬,∨}

Les différents connecteurs correspondent à des constructions logiques essentiellement issues du langage naturel. En réalité ces constructions sont redondantes et un très petit nombre de connecteurs permet d'exprimer tous les autres. Par exemple on voit dans la suite que les trois connecteurs {⊤,¬,∨} peuvent être combinés pour remplacer tous les autres en utilisant les propriétés d'équivalence prouvées plus haut.

Définition de la propriété "être formé uniquement avec des ¬, ∨ et ⊤".

```
Inductive is_reduced : formule ⟶ Prop :=
  Vrai_is_red : is_reduced ⊤
| Non_is_red : forall f,is_reduced f ⟶ is_reduced (¬ f)
| Or_is_red : forall f g,is_reduced f ⟶ is_reduced g ⟶ is_reduced (f ∨ g).
```

Fonction qui remplace les formules contenant les autres connecteurs par des formule réduites équivalentes.

```
Function reduce (f :formule) : formule :=
  match f with
    | ⊤ => ⊤
    | ⊥ => ¬ ⊤
    | ¬ f => ¬ (reduce f)
    | f1 ∨ f2 => (reduce f1) ∨ (reduce f2)
    | f1 ∧ f2 => ¬ (¬ (reduce f1) ∨ ¬ (reduce f2))
    | f1 ⇒ f2 => ¬ (reduce f1) ∨ (reduce f2)
  end.
```

la transformation est toujours équivalente.

```
Lemma reduce_correct : forall f :formule, f ≡ (reduce f).
```

La transformation produit bien une formule réduite, c'est-à-dire ne contenant que les connecteurs {⊤,¬,∨}.

```
Lemma reduce_complete : forall f, is_reduced (reduce f).
```

Chapitre 5

logique_propositionnelle_avec_variables

Ce module formalise la logique propositionnelle (avec variable propositionnelle).

5.1 Le formules propositionnelle avec variables propositionnelles

5.1.1 Définition des formules

Les formules sont définies (par induction) par la grammaire suivante.

```
Inductive formule : Type :=
| Vrai : formule
| Faux : formule
| Var : ℕ ⟶ formule  On désigne les variables par des entiers.
| Non : formule ⟶ formule
| Ou : formule ⟶ formule ⟶ formule
| Et : formule ⟶ formule ⟶ formule
| Implique : formule ⟶ formule ⟶ formule.
```

5.1.2 Exemples de formules sans notation

```
Check Vrai.
Check Faux.
Check (Var 23).
Check (Ou Vrai Vrai).
Check (Ou Faux Vrai).
Check (Ou (Ou Vrai Faux) Vrai).
```

5.1.3 Notations usuelles

```
Notation "⊤" := Vrai.
Notation "⊥" := Faux.
Notation "¬ X" := (Non X).
Notation "X ∨ Y" := (Ou X Y).
Notation "X ∧ Y" := (Et X Y).
```

`Notation "X ⇒ Y" := (Implique X Y).`

Les variables seront également plus lisibles en notant le numéro de la variable en indice de la lettre `X` ('x' majuscule).

`Notation "'X`$_1$`'" := (Var 1).`
`Notation "'X`$_2$`'" := (Var 2).`

5.1.4 Exemples avec les notations usuelles

`Check ⊤.`
`Check ⊥.`
`Check X`$_7$`.`
`Check (⊤ ∨ ⊤).`
`Check (⊤ ∨ X`$_3$`).`
`Check (X`$_9$` ∨ X`$_2$`).`
`Check ((⊤ ∨ X`$_4$`) ∨ ⊤).`

5.2 Interprétation d'une formule (Sémantique)

5.2.1 Définition de l'interprétation d'une formule

Définition de l'interprétation d'une formule. Pour interpréter les variables propositionnelles on a besoin d'une valuation `v` des variables vers les valeurs de vérité (`bool`), ensuite on applique les tables de vérité des feuilles jusqu'à la racine comme le calcul propositionnel.

```
Function interp_def (v :nat ⟶ bool) (f :formule) : bool :=
  match f with
    | ⊤ => table_Vrai
    | ⊥ => table_Faux
    | Var i => v i
    | ¬ f => table_Non (interp_def v f)
    | f1 ∨ f2 => table_Ou (interp_def v f1) (interp_def v f2)
    | f1 ∧ f2 => table_Et (interp_def v f1) (interp_def v f2)
    | f1 ⇒ f2 => table_Implique (interp_def v f1) (interp_def v f2)
  end.
```

5.2.2 quelques exemples

Voir plus bas pour plus d'exemples.

`Eval compute in interp_def (fun x => true) (Implique ⊤ (Ou ⊤ ⊥)).` → true

`Eval compute in interp_def (fun x => false) (Implique ⊤ (Ou ⊤ ⊥)).` → true

`Eval compute in interp_def (fun x => false) (Implique ⊤ (Ou ⊥ ⊥)).` → false

`Definition val_X`$_1$`_true := (fun x => match x with 1 => true | _ => false end).`

`Definition val_X`$_1$`_false := (fun x => match x with 1 => false | _ => false end).`

`Eval compute in interp_def val_X`$_1$`_true (Implique ⊤ (Ou ⊥ X`$_1$`)).` → true

`Eval compute in interp_def val_X`$_1$`_false (Implique ⊤ (Ou ⊥ X`$_1$`)).` → false

Version optimisée qui n'évalue la sous-formule de droite que si nécessaire.

```
Function interp (v :nat ⟶ bool) (f :formule) : bool :=
  match f with
    | ⊤ => true
    | ⊥ => false
    | Var i => v i
    | ¬f => if interp v f then false else true
    | f1 ∨ f2 => if interp v f1 then true else interp v f2
    | f1 ∧ f2 => if interp v f1 then interp v f2 else false
    | f1 ⇒ f2 => if interp v f1 then interp v f2 else true
  end.
```

5.2.3 Preuve de correction de la version optimisée.

```
Lemma interp_correct : forall I :ℕ ⟶ bool,forall f :formule, interp_def I f = interp I f.
```

5.2.4 Exemples d'interprétations

```
  Definition v1 (n :nat) : bool :=
    match n with
      | 1 => true
      | 2 => false
      | 3 => true
      | _ => false
    end.
  Definition v2 (n :nat) : bool :=
    match n with
      | 1 => false
      | 2 => true
      | 3 => true
      | _ => false
    end.
  Eval compute in interp_def v1 (Implique Vrai (Ou Vrai Faux)).  → true
  Eval compute in interp_def v2 (Implique Vrai (Ou Vrai Faux)).  → true
  Eval compute in interp_def v1 (Implique Vrai (Ou Faux Faux)).  → false
  Eval compute in interp_def v2 (Implique Vrai (Ou Faux Faux)).  → false
  Eval compute in interp_def v1 X2.  → false
  Eval compute in interp_def v1 X2.  → false
  Eval compute in interp_def v2 X2.  → true
  Eval compute in interp_def v1 (Implique X1 (Ou Faux Faux)).  → false
  Eval compute in interp_def v2 (Implique X1 (Ou Faux Faux)).  → true
  Eval compute in interp v1 (Implique Vrai (Ou Vrai Faux)).  → true
  Eval compute in interp v2 (Implique Vrai (Ou Vrai Faux)).  → true
  Eval compute in interp v1 (Implique Vrai (Ou Faux Faux)).  → false
  Eval compute in interp v2 (Implique Vrai (Ou Faux Faux)).  → false
```

`Eval compute in interp` v_1 X_2. $\rightarrow$ false

`Eval compute in interp` v_1 X_2. $\rightarrow$ false

`Eval compute in interp` v_2 X_2. $\rightarrow$ true

`Eval compute in interp` v_1 (`Implique` X_1 (`Ou Faux Faux`)). $\rightarrow$ false

`Eval compute in interp` v_2 (`Implique` X_1 (`Ou Faux Faux`)). $\rightarrow$ true

5.3 Conséquence, modèle etc

On applique les définitions de conséquence, modèle etc du chapitre « logique generique » avec les notions de valuation et d'interprétation ci-dessous.

`Definition valuation : Type` := $\mathbb{N} \longrightarrow$ `bool`.

Dans la suite on écrira `interpretation v` φ `b` plutôt que **`interp_def`** `v` φ = `b`.

`Definition interpretation` := `fun v` φ `b => interp_def v` φ `= b`.

Résultat supplémentaire : l'interprétation est décidable. Ce ne sera pas le cas pour la logique des prédicats avec quantificateurs

```
Lemma interpretation_dec :
  forall v φ, interpretation v φ true ∨ interpretation v φ false.
```

Une autre définition pour l'équivalence : l'interprétation des deux formules est toujours identique.

```
Definition equivalent φ1 φ2 :=
  forall I b, (interpretation I φ1 b)<->(interpretation I φ2 b).
```

Preuve d'equivalence entre les deux notions d'équivalence.

`Lemma equivEquivalence :forall` φ_1 φ_2,`(DEFS.equiv` φ_1 φ_2`)` $\leftrightarrow$ `(equivalent` φ_1 φ_2`)`.

5.3.1 Exemple de modèles et de conséquences

`Lemma modele1 : forall v,` $\models$`[v]` $X_1 \vee \neg X_1$.

`Lemma modele2 : forall v,` $\models$`[v] (`$X_1 \vee \neg X_2$`)` $\vee X_2$.

`Lemma conseq1 :` $(X_1 \vee \neg X_2) \wedge X_2 \models X_1$.

5.4 Preuves d'équivalences entre formules

`Lemma eq_implique : forall` φ ψ `: formule,` $\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$.

`Lemma eq_et : forall` φ ψ `: formule,` $(\varphi \wedge \psi) \equiv \neg\ (\neg\varphi \vee \neg\psi)$.

`Lemma eq_not_true :` $\bot \equiv \neg\ \top$.

`Lemma eq_not_false :` $\top \equiv \neg\ \bot$.

5.5 Réduction des connecteurs au sous-ensemble {⊤,¬,∨}

Fonction qui remplace les formules contenant les autres connecteurs par des formule équivalentes.

```
Function reduce (f :formule) : formule :=
  match f with
    | ⊤ => ⊤
    | ⊥ => ¬ ⊤
    | Var i => Var i
    | ¬ f => ¬ (reduce f)
    | f1 ∨ f2 => (reduce f1) ∨ (reduce f2)
    | f1 ∧ f2 => ¬ (¬ (reduce f1) ∨ ¬ (reduce f2))
    | f1 ⇒ f2 => ¬ (reduce f1) ∨ (reduce f2)
  end.
```

```
Lemma reduce_correct :
  forall v, forall f :formule, interp_def v f = interp_def v (reduce f).
```

```
Lemma reduce_equiv : forall f :formule, f ≡ (reduce f).
```

Propriété d'être formé uniquement avec des ¬, ∨ et ⊤ ou Var.

```
Inductive is_reduced : formule ⟶ Prop :=
  Vrai_is_red : is_reduced ⊤
| Var_is_red : forall i, is_reduced (Var i)
| Non_is_red : forall f,is_reduced f ⟶ is_reduced (¬ f)
| Or_is_red : forall f g,is_reduced f ⟶ is_reduced g ⟶ is_reduced (f ∨ g).
```

la fonction `reduce` retourne bien une formule de cette forme.

```
Lemma reduce_complete : forall f, is_reduced (reduce f).
```

5.6 Preuve par réfutation (Utile pour la preuve par tableau plus loin).

```
Lemma conseq_by_contradiction' : forall f1 f2 : formule, f1 ⊨ f2 ⟶ f1 ∧ ¬f2 ⊨ ⊥.
```

Pour prouver f ⊨ g on peut prouver f ∧ ¬g ⊨ ⊥.

```
Lemma conseq_by_contradiction : forall f1 f2 : formule, f1 ∧ ¬f2 ⊨ ⊥ ⟶ f1 ⊨ f2.
```

5.7 Preuve par la méthode des tableaux

5.7.1 Lemmes auxilaires

```
Lemma and_affaiblissement_conseq : forall v f1 f2, ⊨[v] f1∧f2 ⟶ ⊨[v]f1.
```

```
Lemma and_affaiblissement_contr : forall f1 f2, f1 ⊨ ⊥ ⟶ f1∧f2 ⊨ ⊥.
```

```
Lemma Et_sym : forall f1 f2, f1 ∧ f2 ≡ f2 ∧ f1.
```

```
Lemma Et_assoc : forall f1 f2 f3, f1 ∧ (f2 ∧ f3) ≡ (f1 ∧ f2) ∧ f3.
```

```
Function extraction_disjuntion (f F : formule) : option formule :=
  match F with
    | g ∧ F' =>
```

```
      if formule_eq_dec f g then Some F'
      else
        if formule_eq_dec f F' then Some g
        else
          match extraction_disjuntion f F' with
              None => None
            | Some F'' => Some (g ∧ F'')
          end
    | g => if formule_eq_dec f g then Some ⊤ else None
  end.
```

Eval compute in (extraction_disjuntion ($\neg X_1$) ((($X_1 \vee \neg X_2$) $\wedge X_2$) $\wedge \neg X_1 \wedge X_2$)).
Eval compute in (extraction_disjuntion ($\neg X_1$) ((($X_1 \vee \neg X_2$) $\wedge X_2$) $\wedge \neg X_1$)).
Eval compute in (extraction_disjuntion (($X_1 \vee \neg X_2$) $\wedge X_2$) ((($X_1 \vee \neg X_2$) $\wedge X_2$) $\wedge \neg X_1$)).

Lemma Et_et_true : forall f, f ≡ f ∧ ⊤.

Lemma extraction_disjuntion_ok :
  forall f F F', extraction_disjuntion f F = Some F' ⟶ (F ≡ f ∧ F').

5.7.2 Les règles des tableaux

Lemma tableau_Ou :
  forall F f_1 f_2, ((f_1 ∧ F ⊨ ⊥) ∧ (f_2 ∧ F ⊨ ⊥)) ⟶ (f_1 ∨ f_2) ∧ F ⊨ ⊥.

Lemma tableau_Ou' : forall f_1 f_2, (f_1 ⊨ ⊥) ∧ (f_2 ⊨ ⊥) ⟶ (f_1 ∨ f_2) ⊨ ⊥.

Lemma tableau_nonEt :
  forall F f_1 f_2, (¬f_1 ∧ F ⊨ ⊥) ∧ (¬f_2 ∧ F ⊨ ⊥) ⟶ ¬(f_1 ∧ f_2) ∧ F ⊨ ⊥.

Lemma tableau_nonEt' : forall f_1 f_2, (¬f_1 ⊨ ⊥) ∧ (¬f_2 ⊨ ⊥) ⟶ ¬(f_1 ∧ f_2) ⊨ ⊥.

Lemma tableau_implique :
  forall F f_1 f_2, (¬f_1 ∧ F ⊨ ⊥) ∧ (f_2 ∧ F ⊨ ⊥) ⟶ (f_1 ⇒ f_2) ∧ F ⊨ ⊥.

Lemma tableau_implique' : forall f_1 f_2, (¬f_1 ⊨ ⊥) ∧ (f_2 ⊨ ⊥) ⟶ (f_1 ⇒ f_2) ⊨ ⊥.

Lemma tableau_Et : forall F f_1 f_2, f_1 ∧ f_2 ∧ F ⊨ ⊥ ⟶ (f_1 ∧ f_2) ∧ F ⊨ ⊥.

Lemma tableau_nonimplique : forall F f_1 f_2, f_1 ∧ ¬f_2 ∧ F ⊨ ⊥ ⟶ ¬(f_1 ⇒ f_2) ∧ F ⊨ ⊥.

Lemma tableau_nonimplique' : forall f_1 f_2, f_1 ∧ ¬f_2 ⊨ ⊥ ⟶ ¬(f_1 ⇒ f_2) ⊨ ⊥.

Lemma tableau_nonOu : forall F f_1 f_2, ¬f_1 ∧ ¬f_2 ∧ F ⊨ ⊥ ⟶ ¬(f_1 ∨ f_2) ∧ F ⊨ ⊥.

Lemma tableau_nonOu' : forall f_1 f_2, ¬f_1 ∧ ¬f_2 ⊨ ⊥ ⟶ ¬(f_1 ∨ f_2) ⊨ ⊥.

Lemma tableau_ferme_branche : forall F f, (F ∧ f) ∧ ¬f ⊨ ⊥.

Lemma tableau_ferme_branche' : forall F f, f ∧ ¬f ∧ F ⊨ ⊥.

Lemma tableau_ferme_branche'' : forall F f, (f ∧ ¬f) ∧ F ⊨ ⊥.

Lemma p_et_p_eq : forall p, p∧p ≡ p.

5.7.3 Exemples d'application des tableaux.

Lemma conseq1 : ($X_1 \vee \neg X_2$) ∧ X_2 ⊨ X_1.
Proof.

```
  apply conseq_by_contradiction.
  do_Et (X1 ∨ ¬X2) X2.
  do_Ou X1 (¬X2).
  - do_ferme X1.
  - do_ferme X2.
Qed.

Lemma conseq2 : (X1 ∧ X3 ) ∧ (¬X1 ∨ X2) ⊨ ⊥.
Proof.
  do_Ou (¬X1) X2.
  - do_ferme X1.
    Echec
Abort.

Lemma conseq3 : (¬ (X1 ⇒ (X2 ⇒ X1))) ∧ ⊤ ⊨ ⊥.
Proof.
  do_NonImplique X1 (X2 ⇒ X1).
  do_NonImplique X2 X1.
  do_ferme X1.
Qed.

Lemma conseq4 : (¬ (X1 ∨ X2)) ∧ X1 ⊨ ⊥.
Proof.
  do_nonOu X1 X2.
  do_ferme X1.
Qed.

Lemma conseq5 : (¬ (X1 ∧ X2)) ∧ X1 ∧ X2 ⊨ ⊥.
Proof.
  do_nonEt X1 X2.
  - do_ferme X1.
  - do_ferme X2.
Qed.

Lemma conseq_exam_2014 : (¬X2∨¬X3) ⊨ (X2 ∧ X3) ⇒ X1.
Proof.
  apply conseq_by_contradiction.
  do_NonImplique (X2 ∧ X3) (X1).
  do_Et (X2)(X3).
  do_Ou (¬X2) (¬X3).
  - do_ferme X2.
  - do_ferme X3.
Qed.
```

Chapitre 6

logique_generique

Ce chapitre décrit des notions communes à toutes les logiques (hormis la logique propositionnelle sans variable qui est trop simple). On se donne une notion de formule, de valuation et d'interprétation et on définit les notions suivantes :

— modèle
— conséquence
— équivalence
— (in)satisfiabilité
— tautologie.

On étend ensuite ces notion aux environnements, c'est-à-dire aux (multi-)ensembles de formules.

6.1 Les données nécessaires

Une logique consiste en un type des formule et des notions de valuation et d'interprétation.

6.1.1 Un type des formules

```
Parameter formule :Type.
```

6.1.2 Un type des valuation

```
Parameter valuation : Type.
```

6.1.3 Un propriété d'interprétation

Cette propriété prend en paramètre une valuation et attribue une valeur de vérité à une formule.

```
Parameter Inline interpretation : valuation → formule → bool → Prop.
```

La seule propriété exigée est qu'une formule ne peut pas avoir plus d'une interprétation possible.

```
Parameter interpretation_unique :
  ∀ v f b1 b2, interpretation v f b1 → interpretation v f b2 → b1= b2.
```

6.2 Vocabulaire : modèle, conséquence logique, équivalence

6.2.1 modèle

la valuation v est un modèle de la formule f si elle permet d'interpréter f en true. Et cela se note "⊨[v] f", en mathématique le v est en indice du symbole $\models$ ($_v$) .

```
Definition est_modele v f := interpretation v f true.
Notation " '⊨[' v ']' f" := (est_modele v f).
```

6.2.2 Conséquence logique

f_1 a pour conséquence logique f_2 si toute valuation v rendant f_1 vraie rend égalemet f_2 vraie. Autrement tout modèle de f_1 est aussi un modèle de f_2. Et cela se note $f_1 \models f_2$. On peut également trouver : f_1, f_2,...,$f_n \models f$, qui signifie que toute interprétation rendant simultanément toutes les f_i vraies rendent aussi f vraie. Ce qui est donc équivalent en principe à $f_1 \wedge f_2 \wedge ... \wedge f_n \models f$.

```
Definition consequence f1 f2 := ∀ v, (⊨[v] f1) → (⊨[v] f2).
Notation "f1 ⊨ f2" := (consequence f1 f2).
```

6.2.3 Équivalence logique

f_1 est équivalente à f_2 si $f_1 \models f_2$ et $f_2 \models f_1$. Et cela se note $f_1 \equiv f_2$.

```
Definition equiv f1 f2 := (f1 ⊨ f2) ∧ (f2 ⊨ f1).
Notation "f1 ≡ f2" := (equiv f1 f2) (at level 180).
```

6.2.4 Tautologie, formule satisfaisable, insatisfaisable...

Une formule vraie dans toute interprétation est une tautologie, ou formule valide.

```
Definition tautologie (f1 :formule) := ∀ v, ⊨[v] f1.
Notation valide := tautologie (only parsing).
```

Une formule pour laquelle il existe un modèle est dite satifiable, satisfaisable ou réalisable.

```
Definition satisfiable (f1 :formule) := ∃ v, ⊨[v] f1.
Notation realisable := satisfiable (only parsing).
```

Si une telle interprétation n'existe pas, c'est-à-dire si la formule est fausse dans toute interprétation, elle est dite insatisfiable, insatisfaisable, ou encore on dit que c'est une antilogie.

```
Definition insatisfiable (f1 :formule) := ∀ v, interpretation v f1 false.
Notation antilogie := insatisfiable (only parsing).
```

6.3 Environnements

Cette partie définit la notion *d'environnement* c'est-à-dire un ensemble de formule (plus exactement un multi-ensemble : la même formule peut apparaître plusieurs fois. Cette notion sera utilisée par exemple pour la définition de la déduction naturelle, où les jugements sont de la forme $\Gamma \vdash \varphi$, où Γ est un environnement et φ une formule. On définit les notions de modèle, conséquence, satisfiabilité et insatisfiabilité pour un ensemble de formules.

```
Module ENV := MULTISET.MAKELIST(F).
```

Un environnement est un multi-ensemble de formules.

```
Definition env := ENV.t.
```

6.3.1 modèle

```
Definition est_modele_set v Γ := ∀ f, ENV.In f Γ → ⊨[v] f.
Notation " '⊨[[' v ']]' G" := (est_modele_set v G).
```

6.3.2 conséquence

```
Definition consequence_set (Γ : env) f2 := ∀ v, ⊨[[ v ]] Γ → ⊨[v] f2.
Notation "Γ '⊨{}' f" := (consequence_set Γ f).
```

6.3.3 (in)Satisfiabilité

```
Definition satisfiable_set Γ := ∃ v, ⊨[[v]] Γ.
Definition insatisfiable_set Γ := ∀ v, ~(⊨[v] Γ).
```

Un modèle d'une ensemble Γ de formules est également un modèle d'un sous-ensemble de Γ.

```
Lemma est_model_set_subset : ∀ v φ Γ, ⊨[[v]] (ENV.add φ Γ) → ⊨[[v]] Γ.
```

Chapitre 7

logique_predicats

Ce module formalise la logique des prédicats sans quantificateur. Les termes ne contiennent pas de symboles de fonctions, un terme est donc une variable. Les prédicats sont d'arité quelconque.

7.1 Les formules

7.1.1 Les termes

Le type des termes. Les termes ne sont que des variables. La logique propositionnelle inclut en principe les symboles de fonctions mais cela compliquerait la présentation.

```
Inductive terme : Type :=
| TVar : ℕ → terme.
```

7.1.2 Les formules

Le type des formules logique avec prédicats (sans symbole de fonction) et sans quantification. Les noms de prédicats sont représentés par des numéros, un prédicat prend un seul argument : une liste de termes.

```
Inductive formule : Type :=
| Vrai : formule
| Faux : formule
| Var : ℕ → formule
| Non : formule → formule
| Ou : formule → formule → formule
| Et : formule → formule → formule
| Implique : formule → formule → formule
| Pred : ℕ → list terme → formule.
```

Exemples de formules :

`Check (Pred 1 (TVar 1::nil)).` : formule

`Check (Ou (Ou (Pred 1 (cons (TVar 1) (cons (TVar 2) nil))) Faux) Vrai).`

Notations usuelles

```
Notation "⊤" := Vrai.
Notation "⊥" := Faux.
Notation "¬ X" := (Non X).
Notation "X ∨ Y" := (Ou X Y).
Notation "X ∧ Y" := (Et X Y).
Notation "X ⇒ Y" := (Implique X Y).

Notation "'X₁'" := (Var 1).
Notation "'X₂'" := (Var 2).

Notation "'x₁'" := (TVar 1).
Notation "'x₂'" := (TVar 2).

Notation "'p₁' l" := (Pred 1 l).
Notation "'p₂' l" := (Pred 2 l).
```

Exemples avec notations

```
Check (Var 7).
Check X₇.
Check (Pred 1 (x₁::nil)).
Check (p₁ [x₁ , x₂ , x₃]).
Check (p₁ [x₁ , x₂ , x₃]).
```

termes : `Check x₇.`

```
Scheme Equality for terme.
```

Décision de l'égalité sur les formules

```
Function formule_beq (φ ψ :formule) {struct φ} : bool :=
  match φ,ψ with
    | ⊤,⊤ ⇒ true
    | ⊥,⊥ ⇒ true
    | Var p,Var q ⇒ beq_nat p q
    | φ₁ ⇒ φ₂, ψ₁ ⇒ ψ₂ ⇒ formule_beq φ₁ ψ₁ && formule_beq φ₂ ψ₂
    | φ₁ ∧ φ₂,ψ₁ ∧ ψ₂ ⇒ formule_beq φ₁ ψ₁ && formule_beq φ₂ ψ₂
    | φ₁ ∨ φ₂,ψ₁ ∨ ψ₂ ⇒ formule_beq φ₁ ψ₁ && formule_beq φ₂ ψ₂
    | ¬φ,¬ψ ⇒ formule_beq φ ψ
    | (Pred n1 l1) , (Pred n2 l2) ⇒ beq_nat n1 n2 &&
                                    forallb2 terme_beq l1 l2
    | _,_ ⇒ false
  end.

Lemma terme_eq_ok : ∀ t1 t2 :terme, terme_beq t1 t2 = true ↔ t1 = t2.

Lemma terme_eq_true_eq_iff : ∀ l1 l2,
              Forall2 (fun x y : terme ⇒ terme_beq x y = true) l1 l2
              ↔ Forall2 Logic.eq l1 l2.
Lemma formule_eq_okr : ∀ φ ψ :formule, formule_beq φ ψ = true → φ = ψ.
```

```
Lemma formule_eq_okl : ∀ φ ψ :formule, φ = ψ → formule_beq φ ψ = true.
Lemma formule_eq_ok : ∀ φ ψ :formule, φ = ψ ↔ formule_beq φ ψ = true.
Lemma formule_neq_ok : ∀ φ ψ :formule, φ ≠ ψ ↔ formule_beq φ ψ = false.
Lemma formule_eq_dec : ∀ f1 f2 : formule, {f1 = f2} + {f1≠f2}.
```

Pour définir l'interprétation d'une formule nous avons de trois fonctions de valuation : celle pour les propositions (comme pour la logique propositionnelle avec variables), celle pour les termes et celle pour les prédicats. Les prédicats ont leur valeur dans les fonctions n-aires (DxD..xD -> bool). En effet u prédicat p à deux argument doit s'interpréter vers une fonction prenant deux entiers et retournant un booléen. Par exemple le prédicat "<" prend deux entier et retourne `true` si le premier est plus petit que le deuxième.

```
Definition val_propositions :=(nat → bool).
Definition val_termes D :=(terme → D).
Definition val_predicat D :=nat → (list D → bool).

Record Valuation (D :Type) :=
  { predicats : val_predicat D;
    termes : val_termes D;
    propositions : val_propositions }.
```

Définition de l'interprétation d'une formule : on applique les tables de vérité des feuilles jusqu'à la racine.

```
Function interp_def (v :Valuation Z) (f :formule) : bool :=
  match f with
    | ⊤ ⇒ table_Vrai
    | ⊥ ⇒ table_Faux
    | Var i ⇒ (v.(propositions) i)
    | Pred i l ⇒ (v.(predicats) i) (map v.(termes) l)
    | ¬ f ⇒ table_Non (interp_def v f)
    | f1 ∨ f2 ⇒ table_Ou (interp_def v f1) (interp_def v f2)
    | f1 ∧ f2 ⇒ table_Et (interp_def v f1) (interp_def v f2)
    | f1 ⇒ f2 ⇒ table_Implique (interp_def v f1) (interp_def v f2)
  end.
```

On utilisera l'abus de notation suivant : `I[f]` à la place de (`interp_def I f`). Attention toutefois à garder en mémoire qu'une interprétation I ne s'applique pas à une formule mais à une variable, c'est la fonction **interp_def** qui permet de généraliser une interprétation aux formules.

```
Module Exemples.
  Definition v1 (n :nat) : bool :=
    match n with
      | 1 ⇒ true
      | 2 ⇒ false
      | 3 ⇒ true
      | _ ⇒ false
    end.

  Definition v2 (n :nat) : bool :=
    match n with
```

```
    | 1 ⇒ false
    | 2 ⇒ true
    | 3 ⇒ true
    | _ ⇒ false
  end.

Definition is_zero l :=
  match l with
    | 0%Z::_ ⇒ true
    | _ ⇒ false
  end.

Definition vp (n :nat) : (list Z → bool) :=
  match n with
    | 1 ⇒ is_zero
    | _ ⇒ (fun _ ⇒ false)
  end.

Definition vt trm :=
  match trm with
    | TVar i ⇒ (0)%Z
  end.

Definition III :Valuation := {| predicats:=vp; termes:=vt; propositions:=v1|}.
Definition III' :Valuation := {| predicats:=vp; termes:=vt; propositions:=v2|}.

Eval compute in interp_def III (Implique ⊤ (Ou ⊤ ⊥)).
Eval compute in interp_def III' (Implique ⊤ (Ou ⊤ ⊥)).
Eval compute in interp_def III (Implique ⊤ (Ou ⊥ ⊥)).
Eval compute in interp_def III' (p1 [x1]).
Eval compute in interp_def III' (Implique (p1 [x1]) ⊥).
Eval compute in interp_def III' (Implique (p1 [x1]) (Ou ⊥ ⊥)).
```

End EXEMPLES.

7.2 Conséquence, modèle etc

On applique les définitions de conséquence, modèle etc du chapitre avec les notions de valuation et d'interprétation ci-dessous.

```
Definition valuation : Type := @Valuation Z.
Definition interpretation : valuation → formule → bool → Prop :=
  fun v f b ⇒ interp_def v f = b.
```

Résultat supplémentaire : l'interprétation est décidable (ce ne ser pas le cas pour la logique des prédicats avec quantificateurs

```
Lemma interpretation_dec : ∀ v f,
                           interpretation v f true ∨ interpretation v f false.
```

Une autre définition pour l'équivalence : l'interprétation des deux formules est toujours identique.

```
Definition equivalent p1 p2 :=
  ∀ v b, (interpretation v p1 b)<->(interpretation v p2 b).
```

Preuve d'equivalence entre equiv et equivalence.

Lemma equivEquivalence :∀ p1 p2,(DEFS.equiv p1 p2) ↔ (equivalent p1 p2).

Module Exemples_modeles.
Lemma modele1 : ∀ v n, ⊨[v] (Var n) ∨ ¬ (Var n).

Lemma modele2 : ∀ v, ⊨[v] (X_1 ∨ ¬ X_2) ∨ X_2.

End Exemples_modeles.

Module Exemples_conseq.
Lemma conseq1 : (X_1∨¬X_2) ∧ X_2 ⊨ X_1.

End Exemples_conseq.

7.3 Preuves d'équivalences entre formules

Lemma eq_implique : ∀ x y : formule, x ⇒ y ≡ ¬x ∨ y.

Lemma eq_et : ∀ x y : formule, (x ∧ y) ≡ ¬ (¬x ∨ ¬y).

Lemma eq_not : ⊥ ≡ ¬ ⊤.

7.4 Preuve par réfutation (Utile pour la preuve par tableau plus loin).

Lemma conseq_by_contradiction' : ∀ f_1 f_2 : formule, f_1 ⊨ f_2 → f_1 ∧ ¬f_2 ⊨ ⊥.

Pour prouver f ⊨ g on peut prouver f ∧ ¬g ⊨ ⊥ .

Lemma conseq_by_contradiction : ∀ f_1 f_2 : formule, ¬f_2 ∧ f_1 ⊨ ⊥ → f_1 ⊨ f_2.

7.5 Preuve par la méthode des tableaux

7.5.1 Lemmes auxilaires

Lemma and_affaiblissement_conseq : ∀ v f_1 f_2, ⊨[v] f_1∧f_2 → ⊨[v]f_1.

Lemma and_affaiblissement_contr : ∀ f_1 f_2, f_1 ⊨ ⊥ → f_1∧f_2 ⊨ ⊥ .

Lemma Et_sym : ∀ f_1 f_2, f_1 ∧ f_2 ≡ f_2 ∧ f_1.

Lemma Et_assoc : ∀ f_1 f_2 f_3, f_1 ∧ (f_2 ∧ f_3) ≡ (f_1 ∧ f_2) ∧ f_3.

```
Function extraction_disjuntion (f F : formule) : option formule :=
  match F with
    | g ∧ F' ⇒
      if formule_beq f g then Some F'
      else
        if formule_beq f F' then Some g
        else
          match extraction_disjuntion f F' with
              None ⇒ None
            | Some F'' ⇒ Some (g ∧ F'')
```

```
        end
    | g ⇒ if formule_eq_dec f g then Some ⊤ else None
  end.
Eval compute in (extraction_disjuntion (¬X1) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1 ∧ X2)).
Eval compute in (extraction_disjuntion (¬X1) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1)).
Eval compute in (extraction_disjuntion ((X1 ∨ ¬X2) ∧ X2) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1)).
Lemma Et_et_true : ∀ f, f ≡ f ∧ ⊤.
Lemma extraction_disjuntion_ok :
  ∀ f F F', extraction_disjuntion f F = Some F' → (F ≡ f ∧ F').
```

7.5.2 Les règles des tableaux

```
Lemma tableau_Ou : ∀ F f1 f2,
                     (f1 ∧ F ⊨ ⊥)
                     ∧ (f2 ∧ F ⊨ ⊥)
                     → (f1 ∨ f2) ∧ F ⊨ ⊥.
Lemma tableau_Ou' : ∀ f1 f2,
                     (f1 ⊨ ⊥)
                     ∧ (f2 ⊨ ⊥)
                     → (f1 ∨ f2) ⊨ ⊥.
Lemma tableau_nonEt : ∀ F f1 f2,
                        (¬f1 ∧ F ⊨ ⊥) ∧ (¬f2 ∧ F ⊨ ⊥)
                        → ¬(f1 ∧ f2) ∧ F ⊨ ⊥.
Lemma tableau_nonEt' : ∀ f1 f2,
                        (¬f1 ⊨ ⊥) ∧ (¬f2 ⊨ ⊥)
                        → ¬(f1 ∧ f2) ⊨ ⊥.
Lemma tableau_implique : ∀ F f1 f2,
                           (¬f1 ∧ F ⊨ ⊥) ∧ (f2 ∧ F ⊨ ⊥)
                           → (f1 ⇒ f2) ∧ F ⊨ ⊥.
Lemma tableau_implique' : ∀ f1 f2,
                           (¬f1 ⊨ ⊥) ∧ (f2 ⊨ ⊥)
                           → (f1 ⇒ f2) ⊨ ⊥.
Lemma tableau_Et : ∀ F f1 f2, f1 ∧ f2 ∧ F ⊨ ⊥ → (f1 ∧ f2) ∧ F ⊨ ⊥.
Lemma tableau_Et' : ∀ f1 f2, f1 ∧ f2 ⊨ ⊥ → (f1 ∧ f2) ⊨ ⊥.
Lemma tableau_nonimplique : ∀ F f1 f2, f1 ∧ ¬f2 ∧ F ⊨ ⊥ → ¬(f1 ⇒ f2) ∧ F ⊨ ⊥.
Lemma tableau_nonimplique' : ∀ f1 f2, f1 ∧ ¬f2 ⊨ ⊥ → ¬(f1 ⇒ f2) ⊨ ⊥.
Lemma tableau_nonOu : ∀ F f1 f2, ¬f1 ∧ ¬f2 ∧ F ⊨ ⊥ → ¬(f1 ∨ f2) ∧ F ⊨ ⊥.
Lemma tableau_nonOu' : ∀ f1 f2, ¬f1 ∧ ¬f2 ⊨ ⊥ → ¬(f1 ∨ f2) ⊨ ⊥.
Lemma tableau_ferme_branche : ∀ F f, (F ∧ f) ∧ ¬f ⊨ ⊥.
Lemma tableau_ferme_branche' : ∀ F f, f ∧ ¬f ∧ F ⊨ ⊥.
Lemma tableau_ferme_branche'' : ∀ F f, (f ∧ ¬f) ∧ F ⊨ ⊥.
Lemma tableau_ferme_branche''' : ∀ f, f ∧ ¬f ⊨ ⊥.
Lemma p_et_p_eq : ∀ a, a∧a ≡ a.
```

7.5.3 Exemples d'application des tableaux.

```
Module Exemple_Tableaux.

Lemma conseq1 : (X1∨¬X2) ∧ X2 ⊨ X1.
Proof.
  apply conseq_by_contradiction.
  do_Ou X1 (¬X2).
  - do_ferme X1.
  - do_ferme X2.
Qed.

Lemma conseq2 : (X1 ∧ X3 ) ∧ (¬X1 ∨ X2) ⊨ ⊥.
Proof.
  do_Et X1 X3.
  do_Ou (¬X1) X2.
  - do_ferme X1.
  -
Abort.

Lemma conseq3 : (¬ (X1 ⇒ (X2 ⇒ X1))) ⊨ ⊥.
Proof.
  do_NonImplique X1 (X2 ⇒ X1).
  do_NonImplique X2 X1.
  do_ferme X1.
Qed.

Lemma conseq4 : (¬ (X1 ∨ X2)) ∧ X1 ⊨ ⊥.
Proof.
  do_nonOu X1 X2.
  do_ferme X1.
Qed.

Lemma conseq5 : (¬ (X1 ∧ X2)) ∧ X1 ∧ X2 ⊨ ⊥.
Proof.
  do_nonEt X1 X2.
  - do_ferme X1.
  - do_ferme X2.
Qed.

Lemma conseq5' : (¬ ((p2((TVar 1::nil)) ) ∧ X2)) ∧ (p2((TVar 1::nil)) ) ∧ X2 ⊨ ⊥.
Proof.
  do_nonEt (p2((TVar 1::nil)) ) X2.
  - do_ferme (p2((TVar 1::nil)) ).
  - do_ferme X2.
Qed.

Lemma conseq6 : (X1 ∧ (p2((TVar 1::nil)) ) ) ∧ (¬X1 ∨ X2) ⊨ ⊥.
Proof.
  do_Ou (¬X1) X2.
  - do_ferme X1.
  -
Abort.
```

End Exemple_Tableaux.

Chapitre 8

logique_predicats_avec_quantificateurs

Ce module formalise la logique des prédicats avec quantificateur, les prédicats sont n-aires (nombre quelconque d'arguments) mais les termes (arguments des prédicats) ne contiennent pas de symboles de fonctions : ce sont uniquement des variables.

8.1 Les formules

8.1.1 Les termes

Les termes ne sont que des variables. La logique propositionnelle inclut en principe les symboles de fonctions mais cela compliquerait la présentation.

```
Inductive terme : Type :=
| TVar : ℕ → terme.
```

Exemples de termes

```
Check (TVar 7).  : terme
```

8.1.2 Les formules

Le type des formules logique avec prédicats (sans symbole de fonction) avec quantificateur. Les noms de prédicats sont représentés par des numéros, un prédicat prend un seul argument : une liste de termes. Les quantificateurs prennent en argument le numéro de la variable quantifiée.

```
Inductive formule : Type :=
| Vrai : formule
| Faux : formule
| Var : ℕ → formule
| Non : formule → formule
| Ou : formule → formule → formule
| Et : formule → formule → formule
| Implique : formule → formule → formule
| Pred : ℕ → list terme → formule
| FORALL : ℕ → formule → formule
| EXIST : ℕ → formule → formule.
```

Exemples de formules

```
Check (Pred 1 (TVar 1::nil)).  : formule
Check (Ou (Ou (Pred 1 (cons (TVar 1) (cons (TVar 2) nil))) Faux) Vrai).
Check (FORALL 1 (Ou (Ou (Pred 1 (cons (TVar 1) (cons (TVar 2) nil))) Faux) Vrai)).
Check (EXIST 1 (Ou (Ou (Pred 1 (cons (TVar 1) (cons (TVar 2) nil))) Faux) Vrai)).
```

Notations usuelles pour les formules.

```
Notation "⊤" := Vrai.
Notation "⊥" := Faux.
Notation "¬ X" := (Non X).
Notation "X ∨ Y" := (Ou X Y).
Notation "X ∧ Y" := (Et X Y).
Notation "X ⇒ Y" := (Implique X Y).

Notation "'X1'" := (Var 1).
Notation "'X2'" := (Var 2).

Notation "'x1'" := (TVar 1).
Notation "'x2'" := (TVar 2).

Notation "'p1' l" := (Pred 1 l).
Notation "'p2' l" := (Pred 2 l).

Notation "'∀x1' frm" := (FORALL 1 frm).
Notation "'∀x2' frm" := (FORALL 2 frm).

Notation "'∃x1' frm" := (EXIST 1 frm).
Notation "'∃x2' frm" := (EXIST 2 frm).
```

Exemples avec et sans notations :

```
Check (Var 7).
Check X7.
Check (Pred 1 (x1::nil)).
Check (p1 [x1 , x2 , x3]).
Check (p1 [x1 , x2 , x3]).
Check (∀x1 p1 [x1 , x2 , x3]).
Check (∀x1∃x2 p1 [x1 , x2 , x3]).
```

termes : `Check (TVar 7).`
`Check x7.`

`Scheme Equality for terme.`

Décision de l'égalité sur les formules, Attention ce n'est pas la vraie égalité de formule puisque les noms de variables ne sont pas ignorés.

```
Function formule_beq (φ ψ :formule) {struct φ} : bool :=
  match φ,ψ with
```

```
    | ⊤,⊤ ⇒ true
    | ⊥,⊥ ⇒ true
    | Var p,Var q ⇒ beq_nat p q
    | φ1 ⇒ φ2, ψ1 ⇒ ψ2 ⇒ formule_beq φ1 ψ1 && formule_beq φ2 ψ2
    | φ1 ∧ φ2,ψ1 ∧ ψ2 ⇒ formule_beq φ1 ψ1 && formule_beq φ2 ψ2
    | φ1 ∨ φ2,ψ1 ∨ ψ2 ⇒ formule_beq φ1 ψ1 && formule_beq φ2 ψ2
    | ¬φ,¬ψ ⇒ formule_beq φ ψ
    | (Pred n1 l1) , (Pred n2 l2) ⇒ beq_nat n1 n2 &&
                                    forallb2 terme_beq l1 l2
    | FORALL i φ, FORALL j ψ ⇒ beq_nat i j && formule_beq φ ψ
    | EXIST i φ, EXIST j ψ ⇒ beq_nat i j && formule_beq φ ψ
    | _,_ ⇒ false
  end.

Lemma terme_eq_ok : ∀ t1 t2 :terme, terme_beq t1 t2 = true ↔ t1 = t2.

Lemma terme_eq_true_eq_iff : ∀ l1 l2,
              Forall2 (fun x y : terme ⇒ terme_beq x y = true) l1 l2
              ↔ Forall2 Logic.eq l1 l2.
Lemma formule_eq_okr : ∀ φ ψ :formule, formule_beq φ ψ = true → φ = ψ.

Lemma formule_eq_okl : ∀ φ ψ :formule, φ = ψ → formule_beq φ ψ = true.

Lemma formule_eq_ok : ∀ φ ψ :formule, φ = ψ ↔ formule_beq φ ψ = true.

Lemma formule_neq_ok : ∀ φ ψ :formule, φ ≠ ψ ↔ formule_beq φ ψ = false.

Lemma formule_eq_dec : ∀ f1 f2 : formule, {f1 = f2} + {f1≠f2}.
```

8.2 Définition de l'interprétation d'une formule

8.2.1 Valuation

Pour interpréter une formule on a besoin des éléments suivants :

- une valuation `I` des variables propositionnelles,
- un domaine d'interprétation pour les termes (`D`),
- une valuation pour les termes, c'est-à-dire les variables de terme puisqu'on n'a pas de symbole de fonction. La valuation `interp_t` prend donc un numéro de variable de terme en argument et retourne un élément de `D`.
- une valuation pour les prédicats. Un prédicat est désigné par un entier, la valuation `interp_p` prend donc un entier et retourne l'interprétation du prédicat correspondant : une fonction de DxDxD... -> bool, où D est le domaine d'interprétation des terme.
  Par ailleurs on aura besoin de s'assurer que le domaine d'interprétation des termes est non vide, on se donne donc un témoin.

```
Record Valuation {D :Type} :=
  { vpredicat : nat → (list D → bool);
    vterme : terme → D;
    vproposition : nat → bool;
    temoin :D }.
```

8.2.2 Substitution dans la valuation des termes

Ajout (où modification) de la valeur d'une variable dans une interprétation. Utile pour l'interprétation des quantificateurs. On se place dans Z pour écrire la fonction.

```
Definition subst_in_interp interp i (n :Z) :=
  {| vpredicat:= interp.(vpredicat) ;
     vproposition:=interp.(vproposition);
     temoin:=interp.(temoin);

      On retourne n si Var i est demandée, sinon on passe la main à l'ancienne fonction.

     vterme:= fun trm:terme ⇒
                 let '(TVar j) := trm in
                 if beq_nat j i then n
                 else (interp.(vterme)) trm |}.
Notation "I [ i <- x ]" := (subst_in_interp I i x) (at level 89).
```

8.2.3 Interprétation d'une formule

La sémantique ne peut plus se définir comme une fonction car elle ne peut pas être le résultat d'un calcul (comment *calculer* la valeur de ∀x∈ℕ,P(x) si le domaine de x est infini).

On définit donc l'interprétation comme une relation entre une formule, une interprétation et un booléen. On prouvera plus bas que cette relation est "fonctionnelle" en utilisant le tiers-exclu. Par ailleurs on prend Z comme domaine d'interprétation des termes mais cette définition est valable pour n'importe quel ensemble non vide.

```
Inductive interp_def (I :@Valuation Z) : formule → bool → Prop :=
| I_Vrai : interp_def I ⊤ table_Vrai
| I_Faux : interp_def I ⊥ table_Faux
| I_Var : ∀ i b, I.(vproposition) i = b → interp_def I (Var i) b
| I_Pred : ∀ i l b, I.(vpredicat) i (map I.(vterme) l) = b
                        → interp_def I (Pred i l) b
| I_Non : ∀ frm b1 b, interp_def I frm b1 →
                          negb b1 = b →
                          interp_def I (¬ frm) b
| I_Ou : ∀ f1 f2 b1 b2 b, interp_def I f1 b1
                                → interp_def I f2 b2
                                → orb b1 b2 = b
                                → interp_def I (f1 ∨ f2) b
| I_Et : ∀ f1 f2 b1 b2 b, interp_def I f1 b1
                                → interp_def I f2 b2
                                → andb b1 b2 = b
                                → interp_def I (f1 ∧ f2) b
| I_Implique : ∀ f1 f2 b1 b2 b, (interp_def I f1 b1)
                                      → (interp_def I f2 b2)
                                      → implb b1 b2 = b
                                      → (interp_def I (f1 ⇒ f2) b)
| I_Forall : ∀ f1 n, (∀ x :Z, interp_def (subst_in_interp I n x) f1 true)
                          → interp_def I (FORALL n f1) true
```

```
| I_NotForall : ∀ f1 n, (∃ x :Z, interp_def (subst_in_interp I n x) f1 false)
                          → interp_def I (FORALL n f1) false
| I_Exist : ∀ f1 n, (∃ x :Z, interp_def (subst_in_interp I n x) f1 true)
                        → interp_def I (EXIST n f1) true
| I_NotExist : ∀ f1 n, (∀ x :Z, interp_def (subst_in_interp I n x) f1 false)
                          → interp_def I (EXIST n f1) false.
```

On utilisera l'abus de notation suivant : `I[f]->b` à la place de (`interp_def I g b`). Attention toutefois à garder en mémoire qu'une interprétation I ne s'applique pas à une formule mais à une variable, c'est la fonction `interp_def` qui permet de généraliser une interprétation aux formules.

```
Notation "I [ f ]-> b" := (interp_def I f b) (at level 50).
```

8.2.4 Déterminisme et totalité de l'interprétation d'une formule.

Pour les logique précédente ces deux propriétés étaient des conséquences du fait que l'interprétation était définie par une *fonction*. Maintenant que l'interprétation est une relation il faut démontrer que cette relation correspond en fa it à une fonction (non calculable).

```
Lemma interp_def_det :
  ∀ f1 interp b1,
    interp_def interp f1 b1
    → ∀ b2, interp_def interp f1 b2 → b1 = b2.
```

Ceci introduit l'axiome suivant : ∀ P : Prop, P ∨ ¬ P.

```
Require Import Classical.
```

La relation d'interprétation est totale : pour toute formule et toute interprétation il existe une valeur interprétation booléenne. Pour information cette preuve utilise le tiers-exclu.

```
Lemma interp_def_tot :
  ∀ f1 interp,
    interp_def interp f1 true ∨ interp_def interp f1 false.
```

Corollaire du déterminisme + totalité de l'interpétation.

```
Lemma interp_not_true :
  ∀ f1 interp,
    ¬ interp_def interp f1 true ↔ interp_def interp f1 false.
```

8.2.5 Exemples d'interprétations

```
Module Exemples.
  Definition v1 (n :nat) : bool :=
    match n with
      | 1 ⇒ true
      | 2 ⇒ false
      | 3 ⇒ true
      | _ ⇒ false
    end.

  Definition v2 (n :nat) : bool :=
    match n with
```

```
    | 1 ⇒ false
    | 2 ⇒ true
    | 3 ⇒ true
    | _ ⇒ false
  end.

Definition is_zero l :=
  match l with
    | 0%Z::_ ⇒ true
    | _ ⇒ false
  end.

Definition vp (n :nat) : (list Z → bool) :=
  match n with
    | 1 ⇒ is_zero
    | _ ⇒ (fun _ ⇒ false)
  end.

Definition vt trm :=
  match trm with
    | TVar i ⇒ (0)%Z
  end.

Definition III :Valuation := {| vpredicat:=vp; vterme:=vt; vproposition:=v1; temoin:=0%Z
|}.
Definition III' :Valuation := {| vpredicat:=vp; vterme:=vt; vproposition:=v2; temoin:=0%Z
|}.

Lemma Ex1 : interp_def III (⊤ ⇒ (⊤ ∨ ⊥)) true.

Lemma Ex1' : interp_def III (⊤ ⇒ (⊤ ∨ ⊥)) false.

Lemma Ex2 : interp_def III' (⊤ ⇒ (⊤ ∨ ⊥)) true.

Lemma Ex3 : interp_def III' (p1 [x1]) true.

Lemma Ex4 : interp_def III' (p1 [x1] ⇒ ⊥) false.

Lemma Ex5 : interp_def III' (p1 [x1] ⇒ (⊥ ∧ ⊥)) false.

Lemma Ex5' : interp_def III' (p1 [x1] ⇒ (⊥ ∧ ⊥)) true.
```

End EXEMPLES.

8.3 Conséquence, modèle etc

On applique les définitions de conséquence, modèle etc du chapitre avec les notions de valuation et d'interprétation ci-dessous.

l'interprétation n'est pas nécessairement décidable. En effet l'ensemble sur lequel sont quantifiées les variables peut être infini.

```
Definition valuation : Type := @Valuation Z.
Definition interpretation : valuation → formule → bool → Prop := interp_def.
```

Module EXEMPLES_MODELES.

```
Lemma modele1 : ∀ v n, ⊨[v] Var n ∨ ¬ (Var n).
```

Lemma modele2 : ∀ v, ⊨[v] (X$_1$ ∨ ¬ X$_2$) ∨ X$_2$.

End EXEMPLES_MODELES.

Module EXEMPLES_CONSEQ.

Lemma conseq1 : (X$_1$∨¬X$_2$) ∧ X$_2$ ⊨ X$_1$.

End EXEMPLES_CONSEQ.

8.4 Preuves d'équivalences entre formules

Lemma eq_implique : ∀ x y : formule, x ⇒ y ≡ ¬x ∨ y.

Lemma eq_et : ∀ x y : formule, (x ∧ y) ≡ ¬ (¬x ∨ ¬y).

Lemma eq_not : ⊥ ≡ ¬ ⊤.

Lemma eq_not_exist : ∀ i, ∀ f : formule, (¬ (FORALL i f)) ≡ EXIST i (¬f).

Lemma eq_not_forall : ∀ i, ∀ f : formule, (¬ (EXIST i f)) ≡ FORALL i (¬f).

8.5 Preuve par réfutation (Utile pour la preuve par tableau plus loin).

Lemma conseq_by_contradiction' : ∀ f$_1$ f$_2$: formule, f$_1$ ⊨ f$_2$ → f$_1$ ∧ ¬f$_2$ ⊨ ⊥.

Pour prouver f ⊨ g on peut prouver f ∧ ¬g ⊨ ⊥ .

Lemma conseq_by_contradiction : ∀ f$_1$ f$_2$: formule, ¬f$_2$ ∧ f$_1$ ⊨ ⊥ → f$_1$ ⊨ f$_2$.

8.6 Preuve par la méthode des tableaux

8.6.1 Lemmes auxilaires pour la méthode des tableaux

Lemma and_affaiblissement_conseq : ∀ v f$_1$ f$_2$, v[f$_1$∧f$_2$]->true → v[f$_1$]->true.

Lemma and_affaiblissement_contr : ∀ f$_1$ f$_2$, f$_1$ ⊨ ⊥ → f$_1$∧f$_2$ ⊨ ⊥ .

Lemma Et_sym : ∀ f$_1$ f$_2$, f$_1$ ∧ f$_2$ ≡ f$_2$ ∧ f$_1$.

Lemma Et_assoc : ∀ f$_1$ f$_2$ f$_3$, f$_1$ ∧ (f$_2$ ∧ f$_3$) ≡ (f$_1$ ∧ f$_2$) ∧ f$_3$.

```
Function extraction_disjuntion (f F : formule) : option formule :=
  match F with
    | (g ∧ F') ⇒
      if formule_beq f g then Some F'
      else
        if formule_beq f F' then Some g
        else
          match extraction_disjuntion f F' with
              None ⇒ None
            | Some F'' ⇒ Some (g ∧ F'')
          end
    | g ⇒ if formule_eq_dec f g then Some ⊤ else None
```

```
  end.

Eval compute in (extraction_disjuntion (¬X1) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1 ∧ X2)).
Eval compute in (extraction_disjuntion (¬X1) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1)).
Eval compute in (extraction_disjuntion ((X1 ∨ ¬X2) ∧ X2) (((X1 ∨ ¬X2) ∧ X2) ∧ ¬X1)).

Lemma Et_et_true : ∀ f, f ≡ f ∧ ⊤.

Lemma extraction_disjuntion_ok :
  ∀ f F F', extraction_disjuntion f F = Some F' → (F ≡ f ∧ F').
```

8.6.2 Les règles des tableaux

```
Lemma tableau_Ou : ∀ F f1 f2,
                      ((f1 ∧ F ⊨ ⊥)
                       ∧ (f2 ∧ F ⊨ ⊥))
                      → (f1 ∨ f2) ∧ F ⊨ ⊥.

Lemma tableau_Ou' : ∀ f1 f2,
                      (f1 ⊨ Faux)
                      ∧ (f2 ⊨ Faux)
                      → (f1 ∨ f2) ⊨ Faux.

Lemma tableau_nonEt : ∀ F f1 f2,
                         (¬f1 ∧ F ⊨ ⊥) ∧ (¬f2 ∧ F ⊨ ⊥)
                         → ¬(f1 ∧ f2) ∧ F ⊨ ⊥.

Lemma tableau_nonEt' : ∀ f1 f2,
                         (¬f1 ⊨ ⊥) ∧ (¬f2 ⊨ ⊥)
                         → ¬(f1 ∧ f2) ⊨ ⊥.

Lemma tableau_implique : ∀ F f1 f2,
                            (¬f1 ∧ F ⊨ ⊥) ∧ (f2 ∧ F ⊨ ⊥)
                            → (f1 ⇒ f2) ∧ F ⊨ ⊥.

Lemma tableau_implique' : ∀ f1 f2,
                            (¬f1 ⊨ ⊥) ∧ (f2 ⊨ ⊥)
                            → (f1 ⇒ f2) ⊨ ⊥.

Lemma tableau_Et : ∀ F f1 f2, f1 ∧ f2 ∧ F ⊨ ⊥ → (f1 ∧ f2) ∧ F ⊨ ⊥.

Lemma tableau_Et' : ∀ f1 f2, f1 ∧ f2 ⊨ ⊥ → (f1 ∧ f2) ⊨ ⊥.

Lemma tableau_nonimplique : ∀ F f1 f2, f1 ∧ ¬f2 ∧ F ⊨ ⊥ → ¬(f1 ⇒ f2) ∧ F ⊨ ⊥.

Lemma tableau_nonimplique' : ∀ f1 f2, f1 ∧ ¬f2 ⊨ ⊥ → ¬(f1 ⇒ f2) ⊨ ⊥.

Lemma tableau_nonOu : ∀ F f1 f2, ¬f1 ∧ ¬f2 ∧ F ⊨ ⊥ → ¬(f1 ∨ f2) ∧ F ⊨ ⊥.

Lemma tableau_nonOu' : ∀ f1 f2, ¬f1 ∧ ¬f2 ⊨ ⊥ → ¬(f1 ∨ f2) ⊨ ⊥.

Lemma tableau_ferme_branche : ∀ F f, (F ∧ f) ∧ ¬f ⊨ ⊥.

Lemma tableau_ferme_branche' : ∀ F f, f ∧ ¬f ∧ F ⊨ ⊥.

Lemma tableau_ferme_branche'' : ∀ F f, (f ∧ ¬f) ∧ F ⊨ ⊥.

Lemma tableau_ferme_branche''' : ∀ f, f ∧ ¬f ⊨ ⊥.

Lemma p_et_p_eq : ∀ a, a∧a ≡ a.
```


8.6.3 Les règles pour les quantificateurs

Ces règles sont plus compliquées à exprimer et à démontrer car elle font appelle à la notion de variables liées, libre et fraîches. On commence par définir ces notions et leur propriétés, puis on énonce les règles pour les quantificateurs.

Notions de variables (de terme) fraîches, liées etc.

```
Inductive is_fresh_terme_list (i :nat) : list terme → Prop :=
| isFreshTrmNil : is_fresh_terme_list i nil
| isFreshTrmCons : ∀ j l, is_fresh_terme_list i l → i≠j
                                → is_fresh_terme_list i (TVar j::l).
```

Une variable de terme i est fraîche dans la formule φ si elle n'apparaît nulle part dans φ.

```
Inductive is_fresh i : formule → Prop :=
| isFreetrue : is_fresh i ⊤
| isFreefalse : is_fresh i ⊥
| isFreeV : ∀ j, is_fresh i (Var j)
| isFreeNon : ∀ φ, is_fresh i φ → is_fresh i (¬φ)
| isFreeEt : ∀ φ ψ, is_fresh i φ →is_fresh i ψ → is_fresh i (φ∧ψ)
| isFreeOr : ∀ φ ψ, is_fresh i φ → is_fresh i ψ → is_fresh i (φ∨ψ)
| isFreeImplique : ∀ φ ψ, is_fresh i φ → is_fresh i ψ → is_fresh i (φ ⇒ ψ)
| isFreePred : ∀ p lt, is_fresh_terme_list i lt → is_fresh i (Pred p lt)
| isFreeFORALL : ∀ j φ, is_fresh i φ → i≠j → is_fresh i (FORALL j φ)
| isFreeEXIST : ∀ j φ, is_fresh i φ → i≠j → is_fresh i (EXIST j φ).
```

Une variable i est non-liée dans une formule φ si elle n'a pas d'occurrence dans un quantificateur.

```
Inductive is_not_bound i : formule → Prop :=
| isNotBoundtrue : is_not_bound i ⊤
| isNotBoundfalse : is_not_bound i ⊥
| isNotBoundV : ∀ j, is_not_bound i (Var j)
| isNotBoundNon : ∀ φ, is_not_bound i φ → is_not_bound i (¬φ)
| isNotBoundEt : ∀ φ ψ, is_not_bound i φ →is_not_bound i ψ → is_not_bound i (φ∧ψ)
| isNotBoundOr : ∀ φ ψ, is_not_bound i φ → is_not_bound i ψ → is_not_bound i (φ∨ψ)
| isNotBoundImplique : ∀ φ ψ, is_not_bound i φ
                                → is_not_bound i ψ → is_not_bound i (φ ⇒ ψ)
| isNotBoundPred : ∀ p lt, is_not_bound i (Pred p lt)
| isNotBoundFORALL : ∀ j φ, is_not_bound i φ → i≠j → is_not_bound i (FORALL j φ)
| isNotBoundEXIST : ∀ j φ, is_not_bound i φ → i≠j → is_not_bound i (EXIST j φ).
```

Une variable fraîche est également non-liée

```
Lemma fresh_is_not_bound : ∀ φ i, is_fresh i φ → is_not_bound i φ.
```

On peut changer l'interprétation d'une variable fraîche sans changer l'interprétation (puisqu'elle n'appraît pas dans la formule).

```
Lemma subst_fresh_id : ∀ i φ,
                        is_fresh i φ
                        → ∀ b v x, (subst_in_interp v i x) [φ]-> b
                                        ↔ v [φ]-> b.
```

8.7 Substitution de variable dans une formule

On se donne une opération de substitution de variable par un terme dans une formule.

```
Function substtermvar (n :nat) (v :nat) (fr :formule) {struct fr} : formule :=
  match fr with
      ⊤ ⇒ ⊤
    | ⊥ ⇒ ⊥
    | Var x ⇒ Var x
    | Non x ⇒ Non (substtermvar n v x)
    | Ou x x0 ⇒ Ou (substtermvar n v x) (substtermvar n v x0)
    | Et x x0 ⇒ Et (substtermvar n v x) (substtermvar n v x0)
    | Implique x x0 ⇒ Implique (substtermvar n v x) (substtermvar n v x0)
    | (Pred i l) ⇒
      Pred i (List.map (fun trm' ⇒
                          let '(TVar v') := trm' in
                          if beq_nat v' n then TVar v else TVar v') l)
    | FORALL i fr' ⇒ if beq_nat n i || beq_nat v i then fr else FORALL i (substtermvar n
v fr')
    | EXIST i fr' ⇒ if beq_nat n i || beq_nat v i then fr else EXIST i (substtermvar n v
fr')
  end.

Notation "f [ i <= trm ]" := (substtermvar i trm f) (at level 85).

Eval compute in (substtermvar 1 2 (p1[x1,x1,x3])).
Eval compute in (substtermvar 1 2 (p1[x2,x1,x3])).
Eval compute in (substtermvar 1 2 (∀x1 p1[x1,x1,x3])).
Eval compute in (substtermvar 1 3 ((∀x1 p1[x1,x1,x3]) ∨ ∃x3 p1[x1,x1,x3])).
Eval compute in (substtermvar 1 2 ((∀x1 p1[x1,x1,x3]) ∨ ∃x3 p1[x1,x1,x3])).
Eval compute in (substtermvar 1 2 ((∀x1 p1[x1,x1,x3]) ∨ ∀x3 p1[x1,x1,x3])).

Lemma subst_substterm_pred :
  ∀ n l i v var b,
    (v [substtermvar i var (Pred n l)]-> b)
    ↔ (subst_in_interp v i (vterme v (TVar var)) [Pred n l]-> b).

Lemma substinterp_substvar_eq :
  ∀ i v φ var b (hfreshv :is_not_bound var φ) (hfreshi :is_not_bound i φ),
    ((v [substtermvar i var φ]-> b) ↔ (subst_in_interp v i (v.(vterme) (TVar var)) [φ]->
b)).

Lemma tableau_forall :
  ∀ F f1 i var,
    is_not_bound var f1
    → is_not_bound i f1
    → substtermvar i var f1 ∧ (FORALL i f1) ∧ F ⊨ ⊥
    → (FORALL i f1) ∧ F ⊨ ⊥.

Lemma tableau_exist :
  ∀ F f1 i j,
    is_fresh j f1
    → is_not_bound i f1
```

→ **is_fresh** j F
→ substtermvar i j f_1 ∧ F ⊨ ⊥
→ EXIST i f_1 ∧ F ⊨ ⊥.

Lemma tableau_exist' :
∀ f_1 i j,
is_fresh j f_1
→ **is_not_bound** i f_1
→ substtermvar i j f_1 ⊨ ⊥
→ EXIST i f_1 ⊨ ⊥.

8.7.1 Exemples d'application des tableaux.

Module Exemple_Tableaux.
Lemma conseq1 : (X_1∨¬X_2) ∧ X_2 ⊨ X_1.
Proof.
apply conseq_by_contradiction.
do_Ou X_1 (¬X_2).
- do_ferme X_1.
- do_ferme X_2.
Qed.

Lemma conseq1' : (X_1∨¬(p_1[x_1])) ∧ p_1[x_1] ⊨ X_1.
Proof.
apply conseq_by_contradiction.
do_Ou X_1 (¬p_1[x_1]).
- do_ferme X_1.
- do_ferme (p_1[x_1]).
Qed.

Lemma conseq1'''' : (∃x_1 ((X_1∨¬(p_1[x_1])) ∧ p_1[x_1])) ⊨ X_1.
Proof.
apply conseq_by_contradiction.
do_exist (∃x_1((X_1 ∨ ¬p_1[x_1]) ∧ p_1[x_1])) 9.
do_Et ((X_1 ∨ ¬p_1 [x_9]))(p_1 [x_9]).
do_Ou X_1 (¬p_1[x_9]).
- do_ferme X_1.
- do_ferme (p_1[x_9]).
Qed.

Lemma conseq1''' : p_1[x_3] ∧ (∃x_1 ((X_1∨¬(p_1[x_1])) ∧ p_1[x_1])) ⊨ X_1.
Proof.
apply conseq_by_contradiction.
do_exist (∃x_1((X_1 ∨ ¬p_1 [x_1]) ∧ p_1 [x_1])) 9.
do_Et ((X_1 ∨ ¬p_1 [x_9]))(p_1 [x_9]).
do_Ou X_1 (¬p_1[x_9]).
- do_ferme X_1.
- do_ferme (p_1[x_9]).
Qed.

Lemma conseq1'' : (X_1∨¬(p_1[x_1])) ∧ (∀x_1 p_1[x_1]) ⊨ X_1.

```
Proof.
  apply conseq_by_contradiction.
  do_Ou X1 (¬p1[x1]).
  - do_ferme X1.
  - do_Forall (∀x1(p1 [x1])) 1.
    do_ferme (p1[x1]).
Qed.

Lemma foralneg : (¬∀x1 p1[x1]) ⊨ ∃x2 ¬p1[x2].
Proof.
  apply conseq_by_contradiction.
  rewrite eq_not_exist.
  do_exist (∃x1 ¬(p1 [x1])) 3.
  rewrite eq_not_forall.
  do_Forall (∀x2(¬(¬p1[x2]))) 3.
  do_ferme (¬p1[x3]).
Qed.

Lemma existneg : (¬∃x1 p1[x1]) ⊨ ∀x2 ¬p1[x2].
Proof.
  apply conseq_by_contradiction.
  rewrite eq_not_exist.
  do_exist (∃x2 (¬(¬(p1 [x2])))) 3.
  rewrite eq_not_forall.
  do_Forall (∀x1(¬p1[x1])) 3.
  do_ferme (¬p1[x3]).
Qed.

Lemma conseq2 : (X1 ∧ X3 ) ∧ (¬X1 ∨ X2) ⊨ ⊥.
Proof.
  do_Ou (¬X1) X2.
  - do_ferme X1.
  - Echec
Abort.

Lemma conseq3 : (¬ (X1 ⇒ (X2 ⇒ X1))) ∧ ⊤ ⊨ ⊥.
Proof.
  do_NonImplique X1 (X2 ⇒ X1).
  do_NonImplique X2 X1.
  do_ferme X1.
Qed.

Lemma conseq4 : (¬ (X1 ∨ X2)) ∧ X1 ⊨ ⊥.
Proof.
  do_nonOu X1 X2.
  do_ferme X1.
Qed.

Lemma conseq5 : (¬ (X1 ∧ X2)) ∧ X1 ∧ X2 ⊨ ⊥.
Proof.
  do_nonEt X1 X2.
  - do_ferme X1.
```

```
    - do_ferme X2.
  Qed.

  Lemma conseq5' : (¬ ((p2((TVar 1::nil)) ) ∧ X2)) ∧ (p2((TVar 1::nil)) ) ∧ X2 ⊨ ⊥.
  Proof.
    do_nonEt (p2((TVar 1::nil)) ) X2.
    - do_ferme (p2((TVar 1::nil)) ).
    - do_ferme X2.
  Qed.

  Lemma conseq6 : (X1 ∧ (p2((TVar 1::nil)) ) ) ∧ (¬X1 ∨ X2) ⊨ ⊥.
  Proof.
    do_Ou (¬X1) X2.
    - do_ferme X1.
    - Echec
  Abort.

End Exemple_Tableaux.

Print Assumptions tableau_forall.
Print Assumptions tableau_exist.
Print Assumptions interp_def_tot.
```

Chapitre 9

deduction_naturelle

On redéfinit la notiation $\models$ pour correspondre à la conséquance logique entre une *ensemble* de formule Γ et une formule `f`. Dans les développement précédents $f_1 \models f_2$ correspondait à la conséquence logique entre deux formule.
`Notation "Γ ⊨ f" := (consequence_set Γ f).`

Ce module définit le système de preuve de la déduction naturelle pour la logique propositionnelle. Il établit les propriétés suivantes de ce système :

- correction vis-à-vis de la sémantique des propriétés
- complétude vis-à-vis de la sémantique des propriétés. Cette preuve est directement inspirée de celle Michel Levy dans son polycopié de cours “Introduction à la logique” et dans le livre correspondant.

9.1 Préliminaires

```
Lemma not_ex_contradictoire_2 :
  ∀ Γ,
    ~(∃ n, Var n ∈ Γ ∧ ((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ))
    → ∀ n, (Var n) ∈ Γ → ~((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ).
```

```
Lemma not_ex_contradictoire_3 :
  ∀ Γ,
    ~(∃ n, (Var n) ∈ Γ ∧ ((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ))
    → ∀ n, (¬Var n) ∈ Γ → ¬ (Var n) ∈ Γ.
```

```
Lemma not_ex_contradictoire_4 :
  ∀ Γ,
    ~(∃ n, (Var n) ∈ Γ ∧ ((¬Var n) ∈ Γ ∨(Var n⇒⊥) ∈ Γ))
    → ∀ n, (Var n ⇒ ⊥) ∈ Γ → ¬(Var n) ∈ Γ.
```

9.2 Déduction naturelle : Définition

Définition inductive d’une preuve en deduction naturelle

```
Inductive pf : env →formule → Prop :=
  ax : ∀ Γ φ, φ ∈ Γ → Γ ⊢ φ
| true_ax : ∀ Γ, Γ ⊢ ⊤
```

```
| impe : ∀ Γ φ1 φ2, Γ ⊢ φ1 ⇒ φ2 → Γ ⊢ φ1 → Γ ⊢ φ2
| impi : ∀ Γ φ1 φ2, φ1 :: Γ ⊢ φ2 → Γ ⊢ φ1 ⇒ φ2
| andi : ∀ Γ φ1 φ2, Γ ⊢ φ1 → Γ ⊢ φ2 → Γ ⊢ φ1 ∧ φ2
| ande1 : ∀ Γ φ1 φ2, Γ ⊢ φ1 ∧ φ2 → Γ ⊢ φ1
| ande2 : ∀ Γ φ1 φ2, Γ ⊢ φ1 ∧ φ2 → Γ ⊢ φ2
| noti : ∀ Γ φ1, φ1 :: Γ ⊢ ⊥ → Γ ⊢ ¬ φ1
| note : ∀ Γ φ1 φ2, Γ ⊢ φ1 → Γ ⊢ ¬ φ1 → Γ ⊢ φ2
| ore : ∀ Γ φ1 φ2 φ3,
          Γ ⊢ φ1 ∨ φ2 → φ1::Γ ⊢ φ3 → φ2::Γ ⊢ φ3 → Γ ⊢ φ3
| ori1 : ∀ Γ φ1 φ2, Γ ⊢ φ1 → Γ ⊢ φ1 ∨ φ2
| ori2 : ∀ Γ φ1 φ2, Γ ⊢ φ2 → Γ ⊢ φ1 ∨ φ2
| fale : ∀ Γ φ1, (¬ φ1) :: Γ ⊢ ⊥ → Γ ⊢ φ1
where " X ⊢ Y" := (pf X Y).
```

9.2.1 Exemples de preuve en déduction naturelle

Certaines de ces preuves sont réutilisées plus bas. On utilise une tactic dédiée pour la règle `ax` : tax. Cette tactique applique `ax` puis essaie de prouver sa prémisse `p` ∈ Γ automatiquement.

```
Module Exemple_DN.
  Lemma ex_DN_1 : pf ([X1]) X1.
    tax.
  Qed.

  Lemma ex_DN_2 : [X1 , X2] ⊢ X1.
    tax.
  Qed.

  Lemma ex_DN_3 : [X2 , X1] ⊢ X1.
    tax.
  Qed.
```

Cette propriété est connue sous le nom « Tiers exclu ». Le tiers exclu est prouvable en déduction naturelle pour toute formule `A`.

```
  Lemma a_or_nota : ∀ A Γ, Γ ⊢ A∨¬A.
  Proof.
    intros A Γ.
    apply fale.
    apply note with (A∨¬A).
    - apply ori2.
      apply noti.
      apply note with (A∨¬A).
      + apply ori1.
        tax.
      + tax.
    - tax.
  Qed.
End Exemple_DN.
```

9.3 Propriétés remarquables de la déduction naturelles

Ces propriétés sont nécessaires à la preuve de complétude de la déduction naturelle.

9.3.1 Compatibilité de la preuve avec la notion d'égalité sur les environnements.

Lemma equiv_gamma : $\forall\ \Gamma\ \Gamma'\ \mathtt{f}$, ENV.eq $\Gamma\ \Gamma' \to \Gamma \vdash \mathtt{f} \to \Gamma' \vdash \mathtt{f}$.

Le morphisme associé

```
Add Parametric Morphism : pf
    with signature ENV.eq ==> Logic.eq ==> iff as pf_morphism.
```

9.3.2 Règles supplémentaires déductible des règles de base

Lemma or_imp : $\forall$ B C Γ, $((B \Rightarrow \bot) \Rightarrow C)\ ::\ \Gamma \vdash (B \vee C)$.

Lemma weakening : $\forall\ \Gamma\ \varphi\ \psi$, $\Gamma \vdash \psi \to \varphi\ ::\ \Gamma \vdash \psi$.

Lemma impe_impi_add : $\forall\ \Gamma\ \varphi\ \varphi'\ \psi$, $\varphi\ ::\ \Gamma \vdash \psi \to \varphi'\ ::\ \Gamma \vdash \varphi \to \varphi'\ ::\ \Gamma \vdash \psi$.

Lemma impe_impi : $\forall\ \Gamma\ \varphi\ \psi$, $\varphi\ ::\ \Gamma \vdash \psi \to \Gamma \vdash \varphi \to \Gamma \vdash \psi$.

Lemma impe_impi_add_double : $\forall\ \Gamma\ \varphi\ \varphi'\ \varphi''\ \psi$,
$\varphi\ ::\ \varphi'\ ::\ \Gamma \vdash \psi \to$
$\varphi''\ ::\ \Gamma \vdash \varphi \to$
$\varphi''\ ::\ \Gamma \vdash \varphi' \to$
$\varphi''\ ::\ \Gamma \vdash \psi$.

Plusieurs des ces lemmes sont en exercices dans la preuve de M. Levy.
Exercie 46 Levy

Lemma and_false_false_or : $\forall$ B C Γ, $((B \wedge C) \Rightarrow \bot)\ ::\ \Gamma \vdash (B \Rightarrow \bot) \vee (C \Rightarrow \bot)$.

Lemma not_not_eq : $\forall\ \Gamma$ A, $\Gamma \vdash A \to \Gamma \vdash \neg(\neg A)$.

Lemma not_not_eq'' : $\forall\ \Gamma$ A, $(\neg(\neg A))\ ::\ \Gamma \vdash A$.

Lemma not_not_eq'''' : $\forall\ \Gamma$ A, $((\neg A) \Rightarrow \bot)\ ::\ \Gamma \vdash A$.

Lemma and_false_false_or' : $\forall$ B C Γ, $(\neg(B \wedge C))\ ::\ \Gamma \vdash (\neg B) \vee (\neg C)$.

Exo 47, partie 1

Lemma not_or_and_not_1 : $\forall$ B C Γ, $(B \vee C \Rightarrow \bot) :: \Gamma \vdash B \Rightarrow \bot$.

Exo 47, partie 2

Lemma not_or_and_not_2 : $\forall$ B C Γ, $(B \vee C \Rightarrow \bot)\ ::\ \Gamma \vdash C \Rightarrow \bot$.

Exo 47, partie 1, avec le not

Lemma not_or_and_not_1' : $\forall$ B C Γ, $(\neg(B \vee C))\ ::\ \Gamma \vdash \neg B$.

Exo 47, partie 2

Lemma not_or_and_not_2' : $\forall$ B C Γ, $(\neg(B \vee C))\ ::\ \Gamma \vdash \neg C$.

exo 48 Levy (1)

Lemma not_imp : $\forall$ B C Δ, $((B \Rightarrow C) \Rightarrow \bot)\ ::\ \Delta \vdash B$.

exo 48 Levy (2)

Lemma not_imp2 : $\forall$ B C Δ, $((B \Rightarrow C) \Rightarrow \bot)\ ::\ \Delta \vdash C \Rightarrow \bot$.

exo 48 Levy (1)

```
Lemma not_imp' : ∀ B C Δ, (¬(B ⇒ C)) :: Δ ⊢ B.
```

exo 48 Levy (2)

```
Lemma not_imp2' : ∀ B C Δ, (¬(B ⇒ C)) :: Δ ⊢ ¬C.
```

9.4 Correction de la déduction naturelle

Preuve de correction de la déduction naturelle vis-à-vis de la sémantique définie dans le chapitre « Logique propositionnelle avec variables ».

```
Lemma soundness : ∀ Γ p, Γ ⊢ p → Γ ⊨ p.
```

9.5 Préliminaires de la Preuve de complétude de la déduction naturelle

9.5.1 Mesure sur une formule seule

```
Function mesure (p :formule){struct p} :nat :=
  match p with
      ⊤ ⇒ 1
    | ⊥ ⇒ 0
    | Var _ =>1
    | ¬ φ2 ⇒ mesure(φ2)+1
    | φ2 ⇒ p2 ⇒ mesure(φ2)+mesure(p2)+1
    | φ2 ∨ p2 ⇒ mesure(φ2)+mesure(p2)+2
    | φ2 ∧ p2 ⇒ mesure(φ2)+mesure(p2)+1
  end.
```

9.5.2 Définition de la mesure sur $\Gamma \vdash \varphi$ en vue de l'induction

Fonctino uxiliaire à la suivante.

```
Definition add_mesure := (fun e acc ⇒ mesure e + acc).
```

Mesure sur un environnement. On utilisera cette définition sur φ : :Γ pour définir la mesure sur $\Gamma \vdash \varphi$. On additionne les mesures de toutes les formules de Γ.

```
Definition gammaMesure (gamma :env) :nat :=
  ENV.fold _ add_mesure gamma 0.
```

Definition de la relation d'ordre sur les environnement induite par la mesure.

```
Definition gammaLt (n m :env) := gammaMesure n < gammaMesure m.
```

Cette relation est bien fondée.

```
Lemma gammaLt_wf : well_founded gammaLt.
```

$\bot$ est le minimum pour cette mesure.

```
Lemma mesure_not_false : ∀ C, C ≠ ⊥ → mesure C > 0.
```

9.5.3 Quelques preuve de compatibilité de la mesure

en vue de permettre les opérations de remplacement (rewrite, symmetry etc)}
L'ordre des formules dans Γ n'est pas significatif.

```
Lemma add_mesure_comm :
  ∀ (k k' : formule) (a : ℕ),
    add_mesure k (add_mesure k' a) = add_mesure k' (add_mesure k a).

Add Morphism add_mesure
    with signature Logic.eq ==> Logic.eq ==> Logic.eq as add_mesure_morphism.

Add Morphism gammaMesure
    with signature ENV.eq ==> Logic.eq as gamma_mesure_morphism.

Lemma transp_add_mesure : ENV.transpose_neqkey nat Logic.eq add_mesure.

Lemma gammaMesure_congru :
  ∀ A Γ, gammaMesure (A :: Γ) = add_mesure A (gammaMesure Γ) .
```

9.5.4 Lemmes auxiliaires sur consequance_set.

```
Lemma conseqset_andl : ∀ {Γ B C}, Γ ⊨ B∧C → Γ ⊨ B.

Lemma conseqset_andr : ∀ {Γ B C}, Γ ⊨ B∧C → Γ ⊨ C.

Lemma conseqset_impe2 : ∀ {Γ B C}, Γ ⊨ B⇒C → B :: Γ ⊨ C.

Lemma conseqset_or_notl : ∀ {Γ B C}, Γ ⊨ B∨C → (B⇒⊥) :: Γ ⊨ C.

Lemma conseqset_lande :
  ∀ {Γ Γ' B C D},
    Γ' == (B∧C) :: Γ → Γ' ⊨ D → (B :: (C :: Γ)) ⊨ D.

Lemma conseqset_neg : ∀ Γ φ, Γ ⊨ ¬φ → (φ :: Γ) ⊨ ⊥.

Lemma conseqset_or_l :
  ∀ {Γ Γ' B C D}, Γ' == (B∨C) :: Γ → Γ' ⊨ D → B :: Γ ⊨ D.

Lemma conseqset_or_r :
  ∀ {Γ Γ' B C D}, Γ' == (B∨C) :: Γ → Γ' ⊨ D → C :: Γ ⊨ D.

Lemma conseqset_imp_l :
  ∀ {Γ Γ' B C D}, Γ' == (B⇒C) :: Γ → Γ' ⊨ D → C :: Γ ⊨ D.

Lemma conseqset_imp_r :
  ∀ {Γ Γ' B C D}, Γ' == (B⇒C) :: Γ → Γ' ⊨ D → (¬B) :: Γ ⊨ D.

Lemma conseqset_notand_l' :
  ∀ {Γ Γ' B C D}, Γ' == (¬(B∧C)) :: Γ → Γ' ⊨ D → (¬B) :: Γ ⊨ D.

Lemma conseqset_notand_r' :
  ∀ {Γ Γ' B C D}, Γ' == (¬(B∧C)) :: Γ → Γ' ⊨ D → (¬C) :: Γ ⊨ D.

Lemma conseqset_notor' :
  ∀ {Γ Γ' B C D}, Γ' == (¬(B∨C)) :: Γ → Γ' ⊨ D → (¬B) :: (¬C) :: Γ ⊨ D.

Lemma conseqset_notimp' :
  ∀ {Γ' B C A}, (¬(B ⇒ C)) :: Γ' ⊨ A → B :: (¬C) :: Γ' ⊨ A.

Lemma conseqset_notimp'' :
  ∀ {Γ' B C A}, (¬¬B) :: Γ' ⊨ A → B :: (¬C) :: Γ' ⊨ A.
```

```
Lemma conseqset_notimp''' :
  ∀ {Γ' B C A}, ((¬B) ⇒⊥) :: Γ' ⊨ A → B :: (¬C) :: Γ' ⊨ A.

Lemma conseqset_notand_l :
  ∀ {Γ Γ' B C D}, Γ' == ((B∧C)⇒⊥) :: Γ → Γ' ⊨ D → (B ⇒ ⊥) :: Γ ⊨ D.

Lemma conseqset_notand_r :
  ∀ {Γ Γ' B C D}, Γ' == ((B∧C)⇒⊥) :: Γ → Γ' ⊨ D → (C ⇒ ⊥) :: Γ ⊨ D.

Lemma conseqset_notor :
  ∀ {Γ Γ' B C D},
    Γ' == ((B∨C)⇒⊥) :: Γ
    → Γ' ⊨ D
    → (B ⇒ ⊥) :: (C ⇒ ⊥) :: Γ ⊨ D.

Lemma conseqset_notimp :
  ∀ {Γ' B C A}, ((B ⇒ C) ⇒ ⊥) :: Γ' ⊨ A → B :: (C ⇒ ⊥) :: Γ' ⊨ A.
```

9.5.5 Notion de formule atomique, litéral, etc

La preuve de complétude de la déduction naturelle fait appel à de nombreux cas de base sur les litéraus et formules atomiques. Les notions nécessaires sont définies dans cette section.

Toutes les formes atomiques de formules, sur-ensemble des litéraux au sens de Levy. les deux définitions ci-après forment une partition de celui-ci.

```
Inductive is_atom : formule → Prop :=
| Atom_var : ∀ n, is_atom (Var n)
| Atom_not : ∀ n, is_atom (¬ (Var n))
| Atom_not2 : ∀ n, is_atom ((Var n) ⇒ ⊥)
| Atom_true : is_atom ⊤
| Atom_not_true : is_atom (¬⊤)
| Atom_not_false : is_atom (¬⊥)
| Atom_false : is_atom ⊥
| Atom_not_false2 : is_atom (⊥ ⇒ ⊥)
| Atom_not_true2 : is_atom (⊤ ⇒ ⊥).
```

Le sous-ensemble des atomes qui ne sont pas des litéraux au sens de Levy. Il s'agit des trois versions atomiques de false.

```
Inductive is_atom_non_literal : formule → Prop :=
| Atomnl_not_true : is_atom_non_literal (¬⊤)
| Atomnl_false : is_atom_non_literal ⊥
| Atomnl_not_true2 : is_atom_non_literal (⊤ ⇒ ⊥).
```

Les littéraux au sens de Levy, un sous ensemble des atomes.

```
Inductive is_literal : formule → Prop :=
| Literal_var : ∀ n, is_literal (Var n)
| Literal_not : ∀ n, is_literal (¬ (Var n))
| Literal_not2 : ∀ n, is_literal ((Var n) ⇒ ⊥)
| Literal_true : is_literal ⊤
| Literal_not_false : is_literal (¬⊥)
| Literal_not_false2 : is_literal (⊥ ⇒ ⊥).

Lemma literal_in_atom : ∀ φ, is_literal φ → is_atom φ.
```

Les atomes sont soit des **is_literal** soit des **is_atom_non_literal**.

```
Lemma atom_in_literal_ln :
  ∀ φ,
    is_atom φ → is_literal φ ∨ is_atom_non_literal φ.
```

la disjonction correspondant aux formules non atomiques.

```
Definition disjunction_non_literal_formula φ :=
  (∃ ψ, ∃ ψ',
                φ = (ψ ∧ ψ')
                ∨ φ = (ψ ∨ ψ')
                ∨ (φ = (ψ ⇒ ψ') ∧ ψ' ≠ ⊥)
                ∨ φ = ((ψ ∧ ψ') ⇒ ⊥)
                ∨ φ = ((ψ ∨ ψ') ⇒ ⊥)
                ∨ φ = ((ψ ⇒ ψ') ⇒ ⊥)
                ∨ φ = (¬(ψ ∧ ψ'))
                ∨ φ = (¬(ψ ∨ ψ'))
                ∨ φ = (¬(ψ ⇒ ψ')))
  ∨ (∃ ψ, φ = (¬ (¬ ψ)) ∨ φ = ((¬ ψ) ⇒ ⊥)).
```

Preuve que **disjunction_non_literal_formula** contient bien les formule non-atomique.

```
Lemma disjunction_non_literal_formula_ok :
  ∀ φ,
    is_atom φ ∨
    disjunction_non_literal_formula φ.
```

Décision de la propriété **is_literal**

```
Lemma is_literal_dec : ∀ φ, is_literal φ ∨ ¬ is_literal φ.
```

9.5.6 Différentes disjonctions sur les termes

La preuve de complétude de la déduction naturelle fait appel à une décomposition par cas sur les termes et sur l'environnement, cette section définit plusieurs disjonctions et propriétés associées.

```
Definition disjunction_formula φ :=
  (∃ ψ, ∃ ψ', φ = (ψ ∧ ψ') ∨ φ = (ψ ⇒ ψ') ∨ φ = (ψ ∨ ψ'))
  ∨ ((¬ is_literal φ) ∧ (∃ ψ, φ = ¬ ψ))
  ∨ is_atom_non_literal φ
  ∨ is_literal φ.

Lemma disjunction_formula_ok : ∀ φ, disjunction_formula φ.

Definition env_disjunction2 Γ :=
  (∃ φ, φ ∈ Γ ∧ disjunction_non_literal_formula φ)
  ∨ (∃ φ, φ ∈ Γ ∧ is_atom_non_literal φ)
  ∨ (∀ ψ, ψ ∈ Γ → is_literal ψ)
  ∨ ENV.Empty Γ.

Lemma env_disjunction_stable_add2 :
  ∀ Γ Γ' n φ, ENV.Add_multiple φ n Γ Γ'
                  → env_disjunction2 Γ
                  → env_disjunction2 Γ'.
```

```
Lemma disjunction_gamma2 :
  ∀ Γ :env, env_disjunction2 Γ.
```

Ceci est plus ou moins la disjonction de Levy utilisé dans la preuve de complétude. Dans le couple $\Gamma \vdash \varphi$, nous avons les cas suivants :

- Soit φ est une formule composée ($\vee$, $\wedge$, $\Rightarrow$, négation d'un non litéral)
- Soit il existe dans Γ une formule composée (légèrement différente : disjunction_non_literal_formula)
- Soit il existe un litéral faux dans Γ (is_atom_non_literal, trois versions , une seule chez Levy)
- Γ ne contient que des litéraux et la formule est atomique (litéral ou faux).

```
Lemma disjunction_gamma_formule :
  ∀ (Γ :env) (φ :formule),
    (∃ ψ ψ', (φ = (ψ ∧ ψ')) ∨ (φ = (ψ ⇒ ψ'))\/ (φ = (ψ ∨ ψ')))
    ∨ ((¬ is_literal φ) ∧ (∃ ψ, φ = ¬ ψ))
    ∨ (∃ ψ, ψ ∈ Γ ∧ disjunction_non_literal_formula ψ)
    ∨ (∃ ψ, ψ ∈ Γ ∧ is_atom_non_literal ψ)
    ∨ ((∀ ψ, ψ ∈ Γ → is_literal ψ) ∧ (is_atom_non_literal φ))
    ∨ ((∀ ψ, ψ ∈Γ → is_literal ψ) ∧ is_literal φ).
```

9.5.7 Notion d'interprétation caractéristique d'un environnement

Définition de l'interpétation caractéristique d'un environnement Γ

```
Definition interp_caracteristique (Γ :ENV.t) : nat → bool := fun n⇒ENV.mem (Var n) Γ.
Definition interp_anticaracteristique (Γ :ENV.t) : nat → bool :=
  fun n⇒negb(orb(ENV.mem (¬Var n) Γ) (ENV.mem (Var n⇒ ⊥) Γ)).

Lemma interp_caracteristique_1_new :
  ∀ Γ,
    (∀ ψ, ψ ∈ Γ → is_literal ψ)
    → ~(∃ n, (Var n)∈ Γ ∧ ((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ))
    → est_modele_set (interp_caracteristique Γ) Γ.

Lemma interp_caracteristique_6_new :
  ∀ Γ n, ¬ (Var n) ∈ Γ → interpretation (interp_caracteristique Γ) (Var n) false.

Lemma interp_caracteristique_5_new :
  ∀ Γ n, ¬ (Var n) ∈ Γ → ¬est_modele (interp_caracteristique Γ) (Var n).

Lemma interp_anticaracteristique_1_new :
  ∀ Γ,
    (∀ ψ, ψ ∈ Γ → is_literal ψ)
    → ~(∃ n, (Var n) ∈ Γ
                  ∧ ((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ))
    → est_modele_set (interp_anticaracteristique Γ) Γ.

Lemma interp_anticaracteristique_6_new :
  ∀ Γ n, ¬ (¬Var n) ∈ Γ → ¬ (Var n ⇒ ⊥) ∈ Γ
              → ⊨[interp_anticaracteristique Γ] (Var n).

Lemma interp_anticaracteristique_not_negb :
  ∀ Γ k,
  interp_def (interp_anticaracteristique Γ) (¬Var k) =
  negb (interp_def (interp_anticaracteristique Γ) (Var k)).
```

9.5.8 Propriétés des environnements ne contenant que des litéraux

```
Lemma ex_listeral_contradictoire_dec_new : ∀ Γ,
  (∃ n, (Var n) ∈ Γ ∧ ((¬Var n) ∈ Γ ∨ (Var n⇒⊥) ∈ Γ))
  ∨ ~(∃ n, (Var n) ∈ Γ ∧ ((¬Var n)∈ Γ ∨ (Var n⇒⊥) ∈ Γ)).
Lemma literaux_conseq_false_new :
  ∀ Γ,
    (∀ ψ, ψ ∈ Γ → is_literal ψ)
    → Γ ⊨ ⊥
    → (∃ n, (Var n) ∈ Γ ∧ ((¬ Var n) ∈ Γ ∨ (Var n ⇒⊥) ∈ Γ)).
Lemma literaux_conseq :
  ∀ Γ k,
    (∀ ψ, ψ ∈ Γ → is_literal ψ)
    → Γ ⊨ (Var k)
    → (∃ n, (Var n) ∈ Γ ∧ ((¬ Var n) ∈ Γ ∨ (Var n⇒ ⊥) ∈ Γ))
      ∨ (Var k) ∈ Γ.
Lemma literaux_conseq_neg :
  ∀ Γ k,
    (∀ ψ, ψ ∈ Γ → is_literal ψ)
    → Γ ⊨ (¬Var k)
    → (∃ n, (Var n) ∈ Γ ∧ ((¬ Var n) ∈ Γ ∨ (Var n⇒ ⊥) ∈ Γ))
      ∨ (¬Var k) ∈ Γ∨ (Var k ⇒ ⊥) ∈ Γ.
```

9.5.9 Preuve de la complétude de la déduction naturelle

Preuve de complétude de la déduction naturelle.

```
Lemma completness : ∀ Γ A, Γ ⊨ A → Γ ⊢ A.
Print Assumptions completness.
```

Chapitre 10

semantique

Ce module formalise la sémantique d'un petit langage impératif. Il n'y a pas de fonction ni de procédure.

La sémantique d'un langage décrit formellement le comportement des expressions et des instructions de ce langage. C'est la "définition" du langage. Pour les expressions il s'agit de définir quelle est la valeur calculée par une expression. Pour une instruction il s'agit de décrire les effets des instructions sur les variables du programme (plus généralement les effets d'un programme peuvent aussi inclure les entrées/sorties mais nous ne considérons pas cela ici).

La sémantique des expression est composée de jugements de la forme $\langle\ \sigma$, e $\rangle \mapsto$ v, où σ est l'enironnement d'exécution qui contient les valeurs des variables du programme, p est l'expression à évaluer et v sa valeur.

La sémantique des instructions sera composée de jugements de la forme $\langle\ \sigma$, NOPE $\rangle \rightsquigarrow \sigma$', où σ est l'enironnement d'exécution avant l'exécution du programme p et σ' est l'environnement après l'exécution de p (les variables ont changé de valeur). On exprime en général cette sémantique sous la forme de règles d'inférence, ce qui s'écrit en Coq par une relation inductive où chaque constructeur correspond à une règle.

10.1 Les expressions

10.1.1 Le type des expressions entières et booléennes

```
Inductive exp : Set :=
| TRUE : exp
| FALSE : exp
| CST : nat ⟶ exp
| VAR : nat ⟶ exp
| PLUS : exp ⟶ exp ⟶ exp
| MINUS : exp ⟶ exp ⟶ exp
| OPP : exp ⟶ exp ⟶ exp
| MULT : exp ⟶ exp ⟶ exp
| DIV : exp ⟶ exp ⟶ exp
| AND : exp ⟶ exp ⟶ exp
| OR : exp ⟶ exp ⟶ exp
| NOT : exp ⟶ exp
```

.

On utilisera les notations suivante pour désigner les expressions de manière plus lisible.

```
Notation "A + B" := (PLUS A B) : Prog_scope.
Notation "A - B" := (MINUS A B) : Prog_scope.
Notation "A * B" := (MULT A B) : Prog_scope.
Notation "A / B" := (DIV A B) : Prog_scope.
Notation "A && B" := (AND A B) : Prog_scope.
Notation "A || B" := (OR A B) : Prog_scope.
Notation "! A" := (NOT A) (at level 45) : Prog_scope.
Notation "'X'" := (VAR 1) : Prog_scope.
Notation "'Y'" := (VAR 2) : Prog_scope.
Notation "'Z'" := (VAR 3) : Prog_scope.
Notation "'T'" := (VAR 4) : Prog_scope.
```

Le domaine d'interprétation des expressions est l'union de l'ensemble des entiers et des booléen. On utlise un type inductif à deux constructeurs.

```
Inductive value : Set :=
  Bool : bool ⟶ value
| Int : nat ⟶ value.
```

10.2 La sémantique opérationnelle à grands pas des expressions

La relation d'interprétation des expressions. Aussi appelée la sémantique des expressions. Afin de présenter la sémantique sous la forme de règles d'inférences, on utilise des barres horizontales. Ces barres n'ont pas de signification (elles sont ignorées par Coq) autre que celle de rappeler la forme des règles d'inférence. Autrement dit la propriété suivante :

$$\frac{\texttt{Ext.binds i v } \sigma \longrightarrow}{\langle\ \sigma, \texttt{VAR i}\ \rangle \mapsto \texttt{v}}$$

est équivalente à `Ext.binds i v` $\sigma \longrightarrow \langle\ \sigma, \texttt{VAR i}\ \rangle \mapsto$ `v`.

Notez que pour des raisons de simplicité on ne définit pas la sémantique de la division. En effet cela nécessiterait de traiter le cas de la division par zéro.

```
Inductive eval_exp (σ :gen_env value) : exp ⟶ value ⟶ Prop :=
| Eval_Var : forall i v,
```

$$\frac{\texttt{Ext.binds i v } \sigma}{\langle\ \sigma\texttt{, VAR i}\ \rangle \mapsto \texttt{v}}$$

```
| Eval_TRUE :
```

$$\frac{}{\langle\ \sigma\texttt{, TRUE}\ \rangle \mapsto \texttt{Bool true}}$$

```
| Eval_FALSE :
```

$$\frac{}{\langle\ \sigma\texttt{, FALSE}\ \rangle \mapsto \texttt{Bool false}}$$

```
| Eval_CST : forall i,
                                                    ---------------------------------
                                                    ⟨ σ , CST i ⟩ ↦ Int i

| Eval_NOT : forall e v v',
                                                                    (⟨ σ , e ⟩ ↦ (Bool v'))
                                                    (v = negb v')
                                                    ---------------------------------------
                                                                ⟨ σ , ! e ⟩ ↦ Bool v

| Eval_AND : forall e1 v1 e2 v2 v,
                                                    ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                                    ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                                    v = andb v1 v2 ⟶
                                                    ---------------------------------
                                                      ⟨ σ , e1 && e2 ⟩ ↦ Bool v

| Eval_OR : forall e1 v1 e2 v2 v,
                                                   ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                                   ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                                   v = orb v1 v2 ⟶
                                                   ---------------------------------
                                                     ⟨ σ , e1 || e2 ⟩ ↦ Bool v

| Eval_PLUS : forall e1 v1 e2 v2 v,
                                                  ⟨ σ , e1 ⟩ ↦ (Int v1) ⟶
                                                  ⟨ σ , e2 ⟩ ↦ (Int v2) ⟶
                                                  v = (v1 +v2)%nat ⟶
                                                  ---------------------------------
                                                  ⟨ σ , e1 + e2 ⟩ ↦ Int v

| Eval_MINUS : forall e1 v1 e2 v2 v,
                                                  ⟨ σ , e1 ⟩ ↦ (Int v1) ⟶
                                                  ⟨ σ , e2 ⟩ ↦ (Int v2) ⟶
                                                  v = (v1 - v2)%nat ⟶
                                                  ---------------------------------
                                                    ⟨ σ , e1 - e2 ⟩ ↦ Int v

| Eval_MULT : forall e1 v1 e2 v2 v,
                                                  ⟨ σ , e1 ⟩ ↦ (Int v1) ⟶
                                                  ⟨ σ , e2 ⟩ ↦ (Int v2) ⟶
                                                  v = (v1 × v2)%nat ⟶
                                                  ---------------------------------
                                                    ⟨ σ , e1 × e2 ⟩ ↦ Int v
```

where " '⟨' A ',' B '⟩' '↦' C " := (**eval_exp** A B C) : Prog_scope.

Exemple de jugement prouvable.

Lemma eval_exp1 : ⟨(ENV.Core.empty _) , CST 1 + CST 2 ⟩ ↦ Int 3.

Preuve du déterminisme de la sémantique des expressions.

```
Lemma determinisme_exp :
  forall σ e v, ⟨σ,e⟩ ↦ v ⟶ forall v', ⟨σ,e⟩ ↦ v' ⟶ v = v'.
```

10.3 Les programmes

10.3.1 Le type des programmes

```
Inductive prog : Set :=
  NOPE : prog
| AFF : nat ⟶ exp ⟶ prog
| SEQ : prog ⟶ prog ⟶ prog
| IFTE : exp ⟶ prog ⟶ prog ⟶ prog
| WHILE : exp ⟶ prog ⟶ prog.
```

Quelques notations pour y voir plus clair.

```
Notation "A ;; B" := (SEQ A B) : Prog_scope.
Notation "N ← B" := (AFF N B) (at level 65) : Prog_scope.
Notation "'X' ← B" := (AFF 1 B) (at level 65) : Prog_scope.
Notation "'Y' ← B" := (AFF 2 B) (at level 65) : Prog_scope.
Notation "'Z' ← B" := (AFF 3 B) (at level 65) : Prog_scope.
Notation "'T' ← B" := (AFF 4 B) (at level 65) : Prog_scope.
Notation "'IF' A 'THEN' B 'ELSE' C" := (IFTE A B C) (at level 200) : Prog_scope.
Notation "'WHILE' A 'DO' B 'DONE'" := (WHILE A B) (at level 71) : Prog_scope.

Module Ex_prog.
```

Exemples de programme ;

```
  Definition prog1 :prog :=
    WHILE TRUE DO
      NOPE ;;
      X ← CST(2) + X;;
      Y ← Y + Z
    DONE.

End Ex_prog.
```

10.3.2 La sémantique (opérationnelle, à grands pas) des programmes

On note `E ≡ F` lorsque E et F sont deux environnements (deux états) équivalents (les variables ont les mêmes valeurs).

La relation d'interprétation des programmes, aussi appelée la sémantique des programmes. Le domaine d'interprétation est `gen_env value` c'est-à-dire qu'un programme est interprété comme une fonction (partielle, certains programmes ne termine pas) des états (ou environnements) vers les états.

```
Inductive eval_prog : prog ⟶ gen_env value ⟶ gen_env value ⟶ Prop :=
```

| Eval_NOPE : forall σ σ',

$$\frac{\sigma \equiv \sigma' \longrightarrow}{\langle\ \sigma\texttt{, NOPE}\ \rangle \rightsquigarrow \sigma'}$$

| Eval_Seq : forall σ p1 σ' p2 σ'',

$$\frac{\langle\ \sigma\texttt{, p1}\ \rangle \rightsquigarrow \sigma' \longrightarrow \langle\ \sigma'\texttt{, p2}\ \rangle \rightsquigarrow \sigma'' \longrightarrow}{\langle\ \sigma\texttt{, p1;;p2}\ \rangle \rightsquigarrow \sigma''}$$

```
| Eval_Aff : forall σ e i v σ',
                                    ⟨ σ, e ⟩ ↦ v ⟶
                                    σ' ≡ σ [ i ← v] ⟶
                                    ──────────────────
                                     ⟨ σ, i ← e ⟩ ⇝ σ'
| Eval_If_then : forall σ e p1 p2 σ',
                                    ⟨ σ, e ⟩ ↦ Bool true ⟶
                                    ⟨ σ, p1 ⟩⇝ σ' ⟶
                                    ──────────────────────────────────
                                              ⟨ σ, (IF e THEN p1 ELSE p2) ⟩ ⇝ σ'
| Eval_If_else : forall σ e p1 p2 σ',
                                    ⟨ σ, e ⟩ ↦ Bool false ⟶
                                    ⟨ σ, p2 ⟩⇝ σ' ⟶
                                    ──────────────────────────────────
                                              ⟨ σ, (IF e THEN p1 ELSE p2) ⟩ ⇝ σ'
| Eval_While_true : forall σ e p σ' σ'',
                                    ⟨ σ, e ⟩ ↦ Bool true ⟶
                                    ⟨ σ, p ⟩ ⇝ σ' ⟶
                                    ⟨ σ', WHILE e DO p DONE ⟩⇝ σ'' ⟶
                                    ──────────────────────────────────
                                    ⟨ σ, WHILE e DO p DONE ⟩ ⇝ σ''

| Eval_While_false : forall σ e p σ',
                                     σ ≡ σ' ⟶
                                     ⟨ σ, e ⟩ ↦ Bool false ⟶
                                     ──────────────────────────
                                       ⟨ σ, WHILE e DO p DONE ⟩ ⇝ σ'

where " '⟨' A ',' B '⟩' '⇝' C " := (eval_prog B A C) : Prog_scope.

Lemma determinisme_prog :
  forall σ p σ', ⟨σ,p⟩ ⇝ σ' ⟶ forall σ'', ⟨σ,p⟩ ⇝ σ'' ⟶ σ' ≡ σ''.
```

Chapitre 11

semantique_avec_routine

Ce module formalise la sémantique d'un petit langage impératif.

```
Reserved Notation "x ; ; y" (at level 70, right associativity).
```

11.1 Les expressions

11.1.1 Le type des expressions entière et booléennes

```
Inductive exp : Set :=
| TRUE : exp
| FALSE : exp
| CST : nat ⟶ exp
| VAR : nat ⟶ exp
| PLUS : exp ⟶ exp ⟶ exp
| MINUS : exp ⟶ exp ⟶ exp
| OPP : exp ⟶ exp ⟶ exp
| MULT : exp ⟶ exp ⟶ exp
| DIV : exp ⟶ exp ⟶ exp
| AND : exp ⟶ exp ⟶ exp
| OR : exp ⟶ exp ⟶ exp
| NOT : exp ⟶ exp.

Notation "A + B" := (PLUS A B) : Prog_scope.
Notation "A - B" := (MINUS A B) : Prog_scope.
Notation "A * B" := (MULT A B) : Prog_scope.
Notation "A / B" := (DIV A B) : Prog_scope.
Notation "A && B" := (AND A B) : Prog_scope.
Notation "A || B" := (OR A B) : Prog_scope.
Notation "! A" := (NOT A) (at level 45) : Prog_scope.
Notation "'X'" := (VAR 1) : Prog_scope.
Notation "'Y'" := (VAR 2) : Prog_scope.
Notation "'Z'" := (VAR 3) : Prog_scope.
Notation "'T'" := (VAR 4) : Prog_scope.
```

Le domaine d'interprétation des expressions

```
Inductive value : Set :=
  Bool : bool ⟶ value
| Int : nat ⟶ value.
```

11.2 La sémantique opérationnelle à grands pas des expressions

Afin de présenter la sémantique sous la forme de règles d'inférences, on introduit la barre horizontal comme une notation qui ne fait rien.

```
Notation "'_____________________' P2" := (P2) (at level 90, P2 at level 200, right associativity, only parsing).
Notation "'_____________________________________' P2" := (P2) (at level 90, P2 at level 200, right associativity, only parsing).
```

La relation d'interprétation des expressions. Aussi appelée la sémantique des expressions.

```
Inductive eval_exp (σ :gen_env value) : exp ⟶ value ⟶ Prop :=
| Eval_Var : forall i v,
                                    Ext.binds i v σ ⟶
                                    ---------------
                                    ⟨ σ, VAR i ⟩↦ v
| Eval_TRUE :
                                    ---------------
                                    ⟨ σ, TRUE ⟩ ↦ Bool true
| Eval_FALSE :
                                    ---------------
                                    ⟨ σ, FALSE ⟩ ↦ Bool false
| Eval_CST : forall i,
                                    ---------------
                                    ⟨ σ, CST i ⟩ ↦ Int i
| Eval_NOT : forall e v v',
                                     ⟨ σ , e ⟩ ↦ (Bool v') ⟶
                                     v = negb v' ⟶
                                     ---------------
                                       ⟨ σ , ! e ⟩ ↦ Bool v
| Eval_AND : forall e1 v1 e2 v2 v,
                                    ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                    ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                    v = andb v1 v2 ⟶
                                    ---------------
                                      ⟨ σ , e1 && e2 ⟩ ↦ Bool v
| Eval_OR : forall e1 v1 e2 v2 v,
                                    ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                    ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                    v = orb v1 v2 ⟶
                                    ---------------
                                      ⟨ σ , e1 || e2 ⟩ ↦ Bool v
| Eval_PLUS : forall e1 v1 e2 v2 v,
```

$$\frac{\langle\ \sigma\ ,\ \texttt{e1}\ \rangle \mapsto (\texttt{Int v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \texttt{e2}\ \rangle \mapsto (\texttt{Int v2}) \longrightarrow \quad \texttt{v} = (\texttt{v1} + \texttt{v2})\%\texttt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \texttt{e1} + \texttt{e2}\ \rangle \mapsto \texttt{Int v}}$$

| Eval_MINUS : forall e1 v1 e2 v2 v,

$$\frac{\langle\ \sigma\ ,\ \texttt{e1}\ \rangle \mapsto (\texttt{Int v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \texttt{e2}\ \rangle \mapsto (\texttt{Int v2}) \longrightarrow \quad \texttt{v} = (\texttt{v1} - \texttt{v2})\%\texttt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \texttt{e1} - \texttt{e2}\ \rangle \mapsto \texttt{Int v}}$$

| Eval_MULT : forall e1 v1 e2 v2 v,

$$\frac{\langle\ \sigma\ ,\ \texttt{e1}\ \rangle \mapsto (\texttt{Int v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \texttt{e2}\ \rangle \mapsto (\texttt{Int v2}) \longrightarrow \quad \texttt{v} = (\texttt{v1} \times \texttt{v2})\%\texttt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \texttt{e1} \times \texttt{e2}\ \rangle \mapsto \texttt{Int v}}$$

```
where " '⟨' A ',' B '⟩' '↦' C " := (eval_exp A B C) : Prog_scope.

Lemma eval_exp1 : ⟨(ENV.Core.empty _) , CST 1 + CST 2 ⟩ ↦ Int 3.

Print eval_exp1.
```

Preuve du déterminisme de la sémantique des expressions.

```
Lemma determinisme_exp :
  forall σ e v, ⟨σ,e⟩ ↦ v ⟶ forall v', ⟨σ,e⟩ ↦ v' ⟶ v = v'.
```

11.3 Les programmes

11.3.1 Le type des programmes

```
Inductive prog : Set :=
  NOPE : prog
| AFF : nat ⟶ exp ⟶ prog
| SEQ : prog ⟶ prog ⟶ prog
| IFTE : exp ⟶ prog ⟶ prog ⟶ prog
| WHILE : exp ⟶ prog ⟶ prog
| CALL : nat ⟶ prog.
```

Quelques notations pour y voir plus clair.

```
Notation "A ;; B" := (SEQ A B) : Prog_scope.
Notation "N ← B" := (AFF N B) (at level 65) : Prog_scope.
Notation "'X' ← B" := (AFF 1 B) (at level 65) : Prog_scope.
Notation "'Y' ← B" := (AFF 2 B) (at level 65) : Prog_scope.
Notation "'Z' ← B" := (AFF 3 B) (at level 65) : Prog_scope.
Notation "'T' ← B" := (AFF 4 B) (at level 65) : Prog_scope.
```

```
Notation "'IF' A 'THEN' B 'ELSE' C" := (IFTE A B C) (at level 200) : Prog_scope.
Notation "'WHILE' A 'DO' B 'DONE'" := (WHILE A B) (at level 71) : Prog_scope.
Notation "'f()'" := (CALL 1) : Prog_scope.
Notation "'g()'" := (CALL 2) : Prog_scope.
Notation "'h()'" := (CALL 3) : Prog_scope.

Module Ex_prog.
  Exemples de programme ;
  Definition prog1 :prog :=
    WHILE TRUE DO
      NOPE ;;
      X ← CST(2) + X;;
      Y ← Y + Z;;
      f()
    DONE.
End Ex_prog.
```

11.3.2 La sémantique (opérationnelle, à grands pas) des programmes

On note $\mathtt{E} \equiv \mathtt{F}$ lorsque E et F sont deux environnements (deux états) équivalents (les variables ont les mêmes valeurs).

```
Notation "E '≡' F" := (Ext.eq E F) (at level 68) : gen_env_scope.
```

La relation d'interprétation des programmes, aussi appelée la sémantique des programmes. Le domaine d'interprétation est **gen_env value** c'est-à-dire qu'un programme est interprété comme une fonction (partielle, certains programmes ne termine pas) des états (ou environnements) vers les états.

```
Inductive eval_prog (π :gen_env prog) : prog ⟶ gen_env value ⟶ gen_env value ⟶
Prop :=
| Eval_NOPE : forall σ σ',
                                        σ ≡ σ' ⟶
                                        ————————————————
                                        ⟨ σ, NOPE, π ⟩ ⇝ σ'
| Eval_Seq : forall σ p1 σ' p2 σ'',
                                        ⟨ σ, p1, π ⟩ ⇝ σ' ⟶
                                        ⟨ σ', p2, π ⟩ ⇝ σ'' ⟶
                                        ————————————————————————
                                        ⟨ σ, p1;;p2, π ⟩ ⇝ σ''
| Eval_Aff : forall σ e i v σ',
                                        ⟨ σ, e ⟩ ↦ v ⟶
                                        σ' ≡ σ [ i ← v] ⟶
                                        ————————————————
                                        ⟨ σ, i ← e, π ⟩ ⇝ σ'
| Eval_If_then : forall σ e p1 p2 σ',
                                        ⟨ σ, e ⟩ ↦ Bool true ⟶
                                        ⟨ σ, p1, π ⟩⇝ σ' ⟶
                                        ————————————————————————————
                                        ⟨ σ, (IF e THEN p1 ELSE p2), π ⟩ ⇝ σ'
```

```
| Eval_If_else : forall σ e p1 p2 σ',
                                        ⟨ σ, e ⟩ ↦ Bool false ⟶
                                        ⟨ σ, p2, π ⟩⇝ σ' ⟶
                                        ──────────────────────────────
                                        ⟨ σ, (IF e THEN p1 ELSE p2), π ⟩ ⇝ σ'
| Eval_While_true : forall σ e p σ' σ'',
                                        ⟨ σ, e ⟩ ↦ Bool true ⟶
                                        ⟨ σ, p, π ⟩ ⇝ σ' ⟶
                                        ⟨ σ', WHILE e DO p DONE, π ⟩⇝ σ'' ⟶
                                        ──────────────────────────────
                                        ⟨ σ, WHILE e DO p DONE, π ⟩ ⇝ σ''

| Eval_While_false : forall σ e p σ',
                                        σ ≡ σ' ⟶
                                        ⟨ σ, e ⟩ ↦ Bool false ⟶
                                        ──────────────────────────────
                                        ⟨ σ, WHILE e DO p DONE, π ⟩ ⇝ σ'
| Eval_Proc : forall σ prc σ' pbody,
                                        Ext.binds prc pbody π ⟶
                                        ⟨ σ, pbody, π ⟩ ⇝ σ' ⟶
                                        ──────────────────────────────
                                        ⟨ σ, CALL prc, π ⟩ ⇝ σ'
```

```
where " '⟨' A ',' B ',' C '⟩' '⇝' D " := (eval_prog C B A D) : Prog_scope.

Lemma determinisme_prog :
  forall π σ p σ', ⟨σ,p,π⟩ ⇝ σ' ⟶ forall σ'', ⟨σ,p,π⟩ ⇝ σ'' ⟶ σ' ≡ σ''.
```

Chapitre 12

semantique_avec_procedure

Ce module formalise la sémantique d'un petit langage impératif.

```
Reserved Notation "x ;; y" (at level 70, right associativity).
```

12.1 Les expressions

12.1.1 Le type des expressions entière et booléennes

```
Inductive exp : Set :=
| TRUE : exp
| FALSE : exp
| CST : nat ⟶ exp
| VAR : nat ⟶ exp
| PLUS : exp ⟶ exp ⟶ exp
| MINUS : exp ⟶ exp ⟶ exp
| OPP : exp ⟶ exp ⟶ exp
| MULT : exp ⟶ exp ⟶ exp
| DIV : exp ⟶ exp ⟶ exp
| AND : exp ⟶ exp ⟶ exp
| OR : exp ⟶ exp ⟶ exp
| NOT : exp ⟶ exp.

Notation "A + B" := (PLUS A B) : Prog_scope.
Notation "A - B" := (MINUS A B) : Prog_scope.
Notation "A * B" := (MULT A B) : Prog_scope.
Notation "A / B" := (DIV A B) : Prog_scope.
Notation "A && B" := (AND A B) : Prog_scope.
Notation "A || B" := (OR A B) : Prog_scope.
Notation "! A" := (NOT A) (at level 45) : Prog_scope.
Notation "'X'" := (VAR 1) : Prog_scope.
Notation "'Y'" := (VAR 2) : Prog_scope.
Notation "'Z'" := (VAR 3) : Prog_scope.
Notation "'T'" := (VAR 4) : Prog_scope.
```

Le domaine d'interprétation des expressions

```
Inductive value : Set :=
  Bool : bool ⟶ value
| Int : nat ⟶ value.
```

12.2 La sémantique opérationnelle à grands pas des expressions

Afin de présenter la sémantique sous la forme de règles d'inférences, on introduit la barre horizontal comme une notation qui ne fait rien.

```
Notation "'_____________________' P2" := (P2) (at level 90, P2 at level 200, right associativity, only parsing).
Notation "'___________________________________' P2" := (P2) (at level 90, P2 at level 200, right associativity, only parsing).
```

La relation d'interprétation des expressions. Aussi appelée la sémantique des expressions.

```
Inductive eval_exp (σ :gen_env value) : exp ⟶ value ⟶ Prop :=
| Eval_Var : forall i v,
                                    Ext.binds i v σ ⟶
                                    ---------------
                                    ⟨ σ, VAR i ⟩↦ v
| Eval_TRUE :
                                    ---------------
                                    ⟨ σ, TRUE ⟩ ↦ Bool true
| Eval_FALSE :
                                    ---------------
                                    ⟨ σ, FALSE ⟩ ↦ Bool false
| Eval_CST : forall i,
                                    ---------------
                                    ⟨ σ, CST i ⟩ ↦ Int i
| Eval_NOT : forall e v v',
                                     ⟨ σ , e ⟩ ↦ (Bool v') ⟶
                                     v = negb v' ⟶
                                     ---------------
                                       ⟨ σ , ! e ⟩ ↦ Bool v
| Eval_AND : forall e1 v1 e2 v2 v,
                                    ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                    ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                    v = andb v1 v2 ⟶
                                    ---------------
                                      ⟨ σ , e1 && e2 ⟩ ↦ Bool v
| Eval_OR : forall e1 v1 e2 v2 v,
                                    ⟨ σ , e1 ⟩ ↦ (Bool v1) ⟶
                                    ⟨ σ , e2 ⟩ ↦ (Bool v2) ⟶
                                    v = orb v1 v2 ⟶
                                    ---------------
                                      ⟨ σ , e1 || e2 ⟩ ↦ Bool v
| Eval_PLUS : forall e1 v1 e2 v2 v,
```

$$\frac{\langle\ \sigma\ ,\ \mathtt{e1}\ \rangle \mapsto (\mathtt{Int\ v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \mathtt{e2}\ \rangle \mapsto (\mathtt{Int\ v2}) \longrightarrow \quad \mathtt{v} = (\mathtt{v1} + \mathtt{v2})\%\mathtt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \mathtt{e1} + \mathtt{e2}\ \rangle \mapsto \mathtt{Int\ v}}$$

| Eval_MINUS : forall e1 v1 e2 v2 v,

$$\frac{\langle\ \sigma\ ,\ \mathtt{e1}\ \rangle \mapsto (\mathtt{Int\ v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \mathtt{e2}\ \rangle \mapsto (\mathtt{Int\ v2}) \longrightarrow \quad \mathtt{v} = (\mathtt{v1} - \mathtt{v2})\%\mathtt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \mathtt{e1} - \mathtt{e2}\ \rangle \mapsto \mathtt{Int\ v}}$$

| Eval_MULT : forall e1 v1 e2 v2 v,

$$\frac{\langle\ \sigma\ ,\ \mathtt{e1}\ \rangle \mapsto (\mathtt{Int\ v1}) \longrightarrow \quad \langle\ \sigma\ ,\ \mathtt{e2}\ \rangle \mapsto (\mathtt{Int\ v2}) \longrightarrow \quad \mathtt{v} = (\mathtt{v1} \times \mathtt{v2})\%\mathtt{nat} \longrightarrow}{\langle\ \sigma\ ,\ \mathtt{e1} \times \mathtt{e2}\ \rangle \mapsto \mathtt{Int\ v}}$$

where " '⟨' A ',' B '⟩' '↦' C " := (**eval_exp** A B C) : Prog_scope.

```
Function eval_exp_list (σ :gen_env value) (le :list exp) (lv :list value) {struct le} :
Prop :=
  match le with
    | nil =>
      match lv with
        | cons v lv' => False
        | nil => True
      end
    | cons e le' =>
      match lv with
        | cons v lv' => eval_exp σ e v ∧ eval_exp_list σ le' lv'
        | nil => False
      end
  end.
```

```
Lemma eval_exp1 : ⟨(ENV.Core.empty _) , CST 1 + CST 2 ⟩ ↦ Int 3.
```

```
Print eval_exp1.
```

Preuve du déterminisme de la sémantique des expressions.

```
Lemma determinisme_exp :
  forall σ e v, ⟨σ,e⟩ ↦ v ⟶ forall v', ⟨σ,e⟩ ↦ v' ⟶ v = v'.
```

```
Corollary deterministe_eval_exp_list : forall σ le lv lv',
eval_exp_list σ le lv
⟶ eval_exp_list σ le lv'
⟶ lv = lv'.
```

12.3 Les programmes

12.3.1 Le type des programmes

```
Inductive prog : Set :=
  NOPE : prog
| AFF : nat ⟶ exp ⟶ prog
| SEQ : prog ⟶ prog ⟶ prog
| IFTE : exp ⟶ prog ⟶ prog ⟶ prog
| WHILE : exp ⟶ prog ⟶ prog
| CALL : nat ⟶ list exp ⟶ prog.
```

Quelques notations pour y voir plus clair.

```
Notation "A ;; B" := (SEQ A B) : Prog_scope.
Notation "N ← B" := (AFF N B) (at level 65) : Prog_scope.
Notation "'X' ← B" := (AFF 1 B) (at level 65) : Prog_scope.
Notation "'Y' ← B" := (AFF 2 B) (at level 65) : Prog_scope.
Notation "'Z' ← B" := (AFF 3 B) (at level 65) : Prog_scope.
Notation "'T' ← B" := (AFF 4 B) (at level 65) : Prog_scope.
Notation "'IF' A 'THEN' B 'ELSE' C" := (IFTE A B C) (at level 200) : Prog_scope.
Notation "'WHILE' A 'DO' B 'DONE'" := (WHILE A B) (at level 71) : Prog_scope.
Notation "'f(' L ')'" := (CALL 1 L) : Prog_scope.
Notation "'g(' L ')'" := (CALL 2 L) : Prog_scope.
Notation "'h(' L ')'" := (CALL 3 L) : Prog_scope.
```

```
Module Ex_prog.
```

Exemples de programme ;

```
  Definition prog1 :prog :=
    WHILE TRUE DO
      NOPE ;;
      X ← CST(2) + X;;
      Y ← Y + Z;;
      f( (X :: (CST(2) + Y)::nil) )
    DONE.
```

```
End Ex_prog.
```

12.3.2 La sémantique (opérationnelle, à grands pas) des programmes

On note `E ≡ F` lorsque E et F sont deux environnements (deux états) équivalents (les variables ont les mêmes valeurs).

```
Notation "E '≡' F" := (Ext.eq E F) (at level 68) : gen_env_scope.
```

La relation d'interprétation des programmes, aussi appelée la sémantique des programmes. Le domaine d'interprétation est `gen_env value` c'est-à-dire qu'un programme est interprété comme une fonction (partielle, certains programmes ne termine pas ou n'ont pas de signification) des états (ou environnements) vers les états.

À la différence des définitions précédentes (sémantiques sans procédure), on n'utilise plus l'équivalence sur les environnement, mais l'égalité stricte. Cela permet de simplifier les preuves sur le pop.

```
Print Scopes.

Inductive eval_prog (π :gen_env (list nat×prog)) : prog ⟶ gen_env value ⟶ gen_env
value ⟶ Prop :=
| Eval_NOPE : forall σ σ',
                                        σ = σ' ⟶
                                        ──────────────────
                                       ⟨ σ, NOPE, π ⟩ ⇝ σ'
| Eval_Seq : forall σ p1 σ' p2 σ'',
                                       ⟨ σ, p1, π ⟩ ⇝ σ' ⟶
                                       ⟨ σ', p2, π ⟩ ⇝ σ'' ⟶
                                       ──────────────────────────────
                                       ⟨ σ, p1;;p2, π ⟩ ⇝ σ''
| Eval_Aff : forall σ e i v σ',
                                       ⟨ σ, e ⟩ ↦ v ⟶
                                        σ' = σ [ i ← v] ⟶
                                        ──────────────────
                                       ⟨ σ, i ← e, π ⟩ ⇝ σ'
| Eval_If_then : forall σ e p1 p2 σ',
                                       ⟨ σ, e ⟩ ↦ Bool true ⟶
                                       ⟨ σ, p1, π ⟩⇝ σ' ⟶
                                        ──────────────────────────────────
                                       ⟨ σ, (IF e THEN p1 ELSE p2), π ⟩ ⇝ σ'
| Eval_If_else : forall σ e p1 p2 σ',
                                       ⟨ σ, e ⟩ ↦ Bool false ⟶
                                       ⟨ σ, p2, π ⟩⇝ σ' ⟶
                                       ──────────────────────────────────
                                       ⟨ σ, (IF e THEN p1 ELSE p2), π ⟩ ⇝ σ'
| Eval_While_true : forall σ e p σ' σ'',
                                       ⟨ σ, e ⟩ ↦ Bool true ⟶
                                       ⟨ σ, p, π ⟩ ⇝ σ' ⟶
                                       ⟨ σ', WHILE e DO p DONE, π ⟩⇝ σ'' ⟶
                                       ──────────────────────────────────
                                       ⟨ σ, WHILE e DO p DONE, π ⟩ ⇝ σ''

| Eval_While_false : forall σ e p σ',
                                       σ = σ' ⟶
                                       ⟨ σ, e ⟩ ↦ Bool false ⟶
                                       ──────────────────────────────────
                                       ⟨ σ, WHILE e DO p DONE, π ⟩ ⇝ σ'

| Eval_Proc : forall σ prc lid le lv σproc σ' σ'' pbody,
                                        Ext.binds prc (lid,pbody) π ⟶
                                        eval_exp_list σ le lv ⟶
                                        pushl lid lv σ = Some σproc ⟶
                                        ⟨ σproc , pbody, π ⟩ ⇝ σ' ⟶
                                        pop_forgetl (List.length lv) σ' = Some σ'' ⟶
                                        ──────────────────────────────────
```

$$\langle\ \sigma\ ,\ \texttt{CALL prc le},\ \pi\ \rangle \rightsquigarrow \sigma''$$

where " '⟨' A ',' B ',' C '⟩' '⇝' D " := (**eval_prog** C B A D) : Prog_scope.

```
Lemma determinisme_prog :
  forall π σ p σ', ⟨σ,p,π⟩ ⇝ σ' ⟶ forall σ'', ⟨σ,p,π⟩ ⇝ σ'' ⟶ σ' = σ''.
```