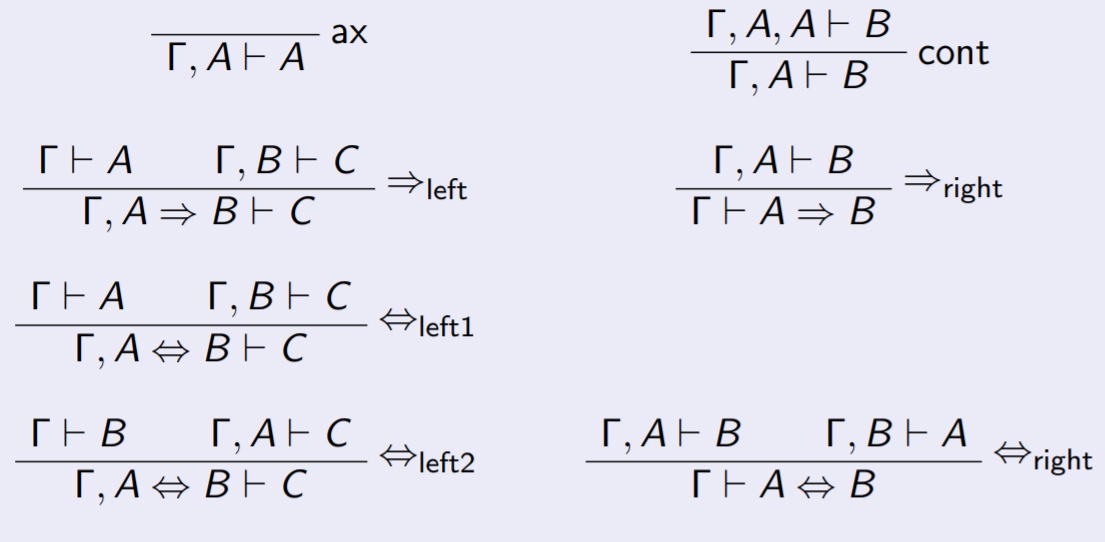
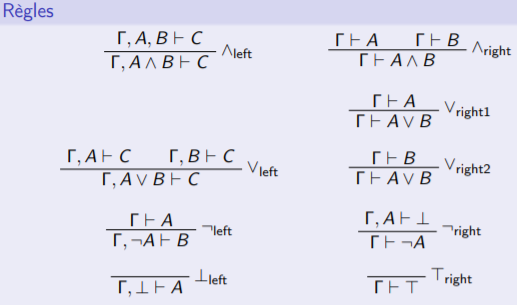
Preuve en logique du premier ordre :

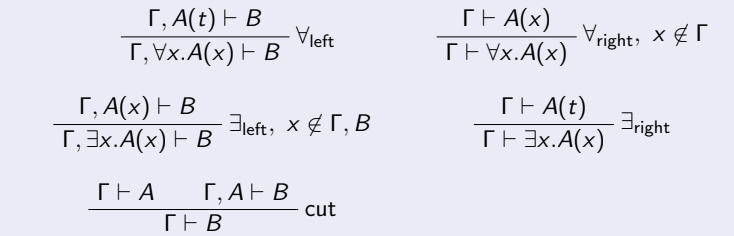
Logique intuitionniste ou constructive

* Une formule est vraie, fausse, ou « on ne sait pas » ;
* Si on ne sait en démontrer la validité, alors « on ne sait pas » ;
* Logique tri-valuée d’une certaine manière ;
* Le « tiers exclu » n’est pas admis dans cette logique.

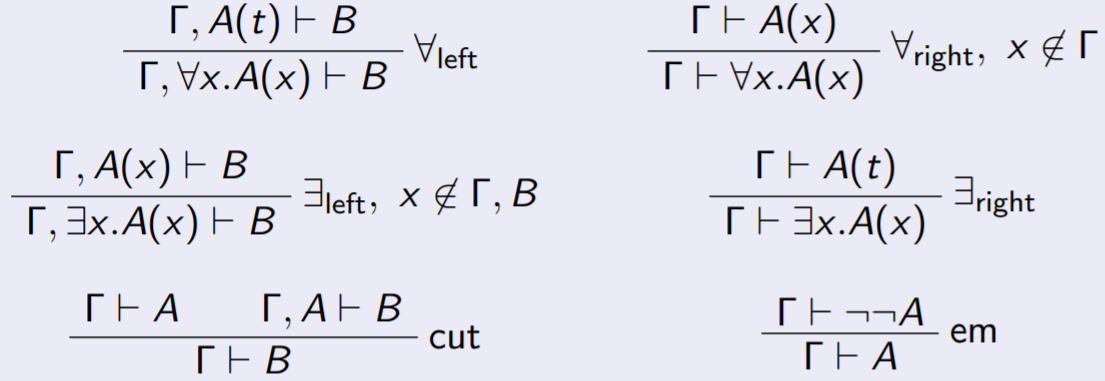
### **Séquents LJ :**



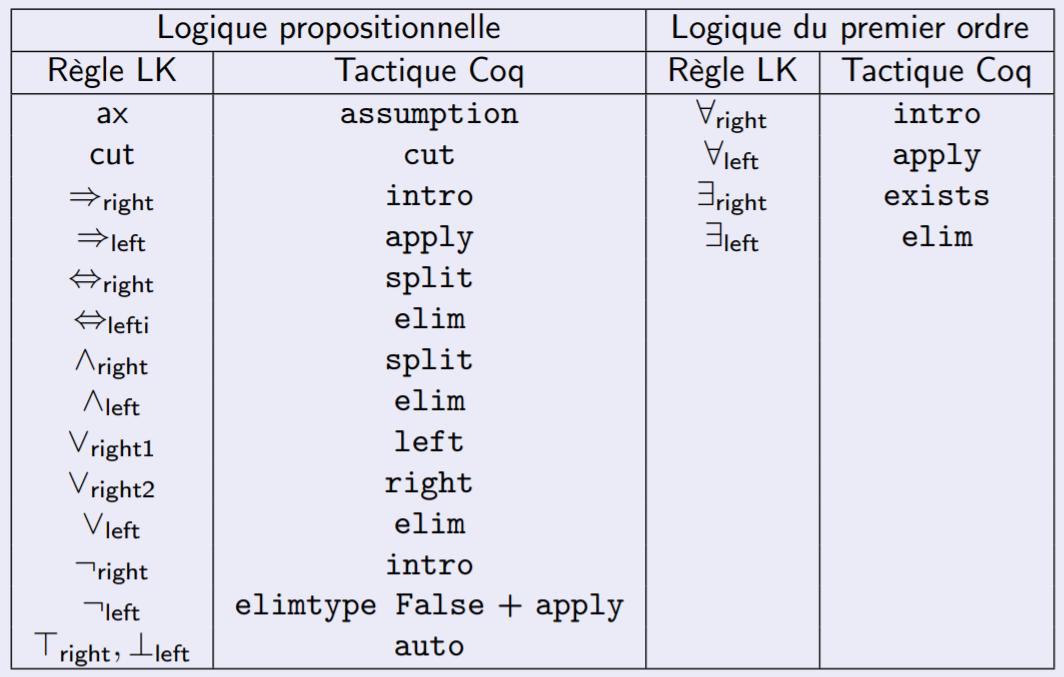


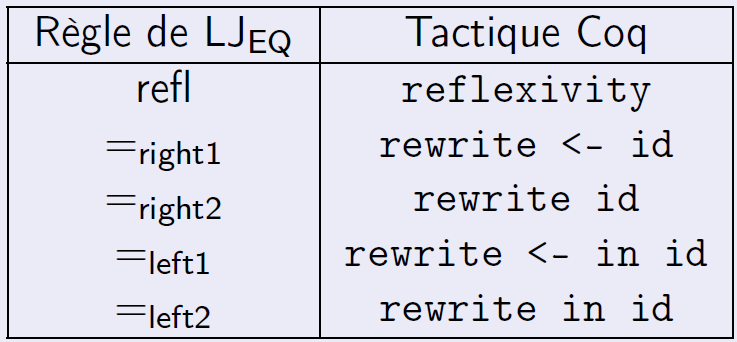


### **LJem :**



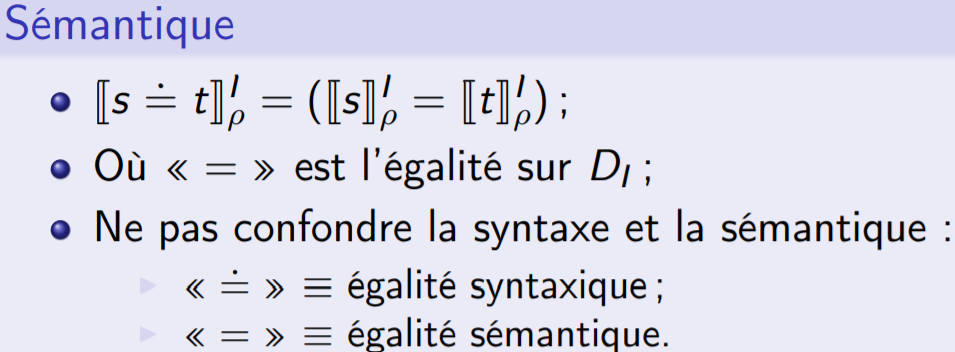
**Correspondance logique → Coq :**



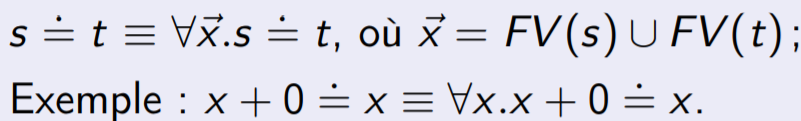


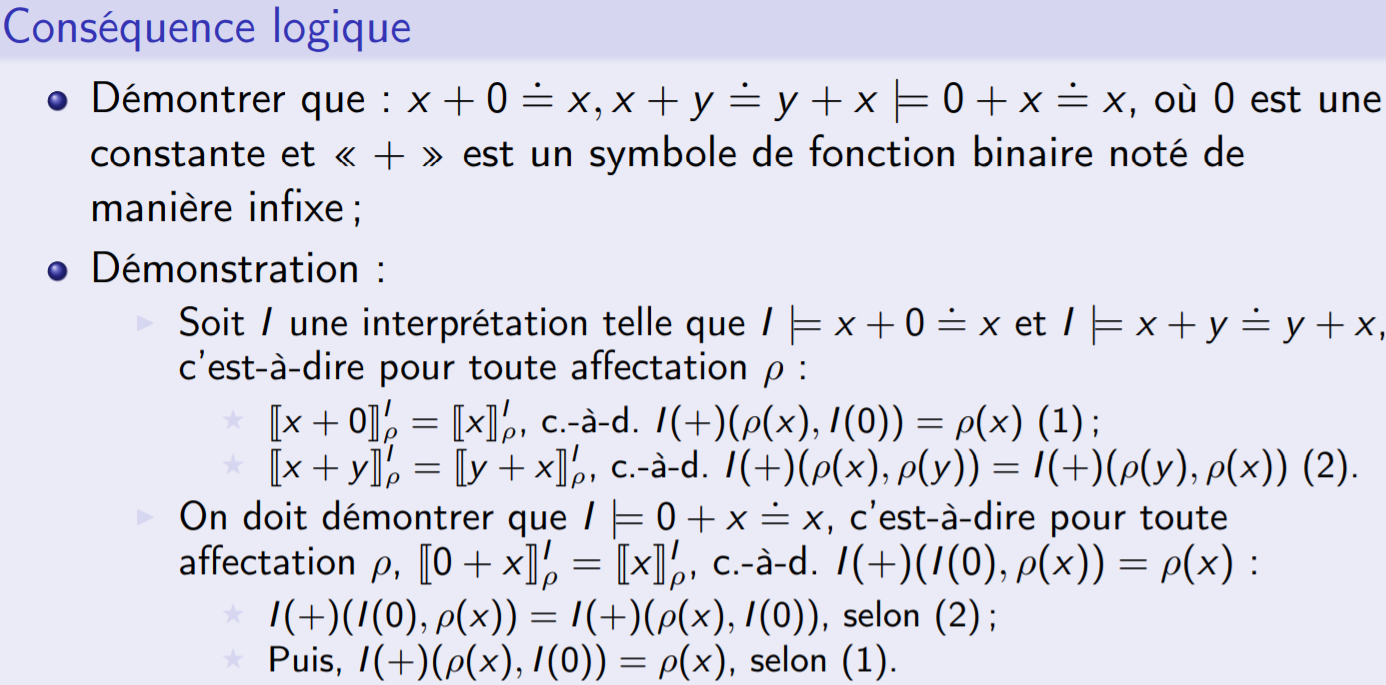
**Preuve en logique équationnelle :**

L’égalité est un prédicat binaire noté de manière infixe : s = t, où s et t sont des termes.

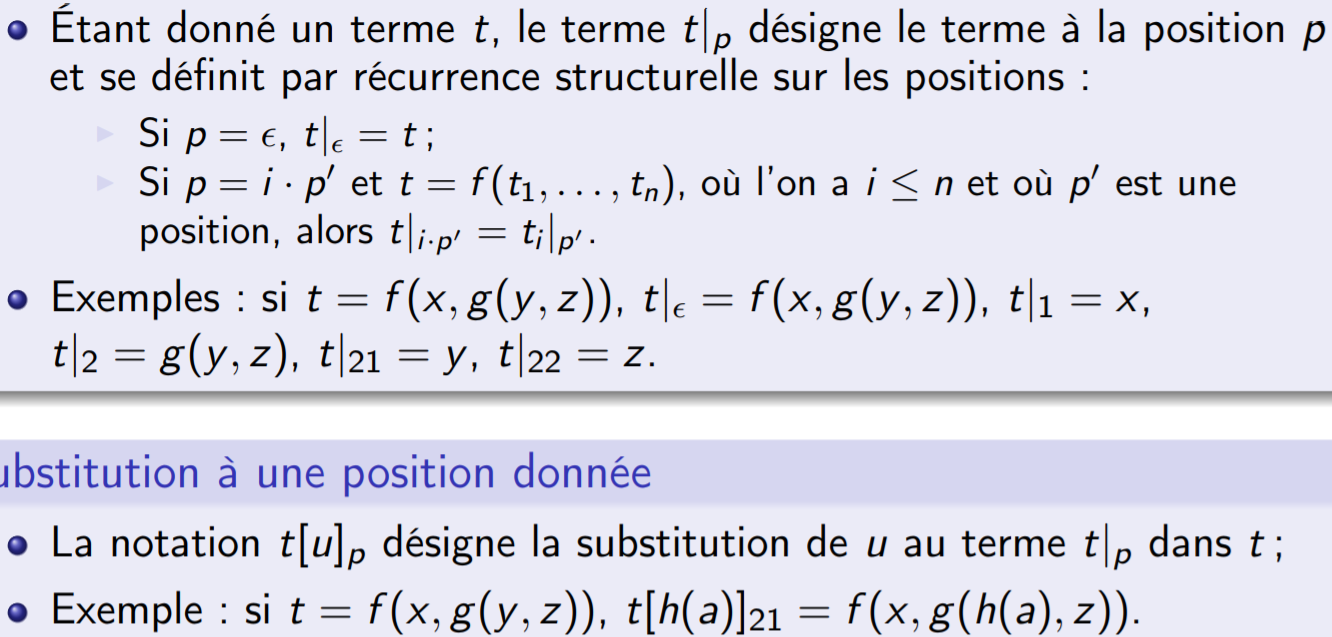


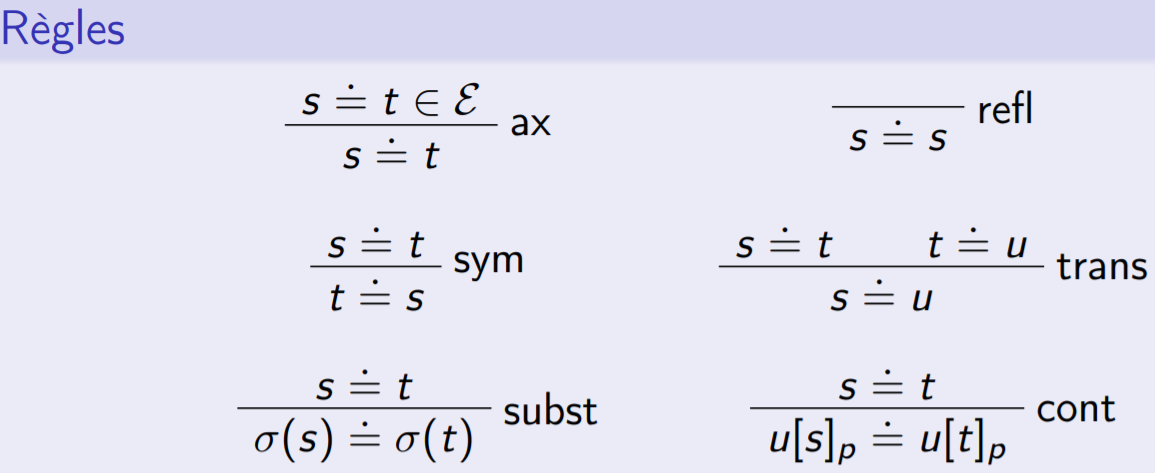
Équation ≡ paire de termes notée s .= t. Les termes s et t ne sont pas forcément clos.

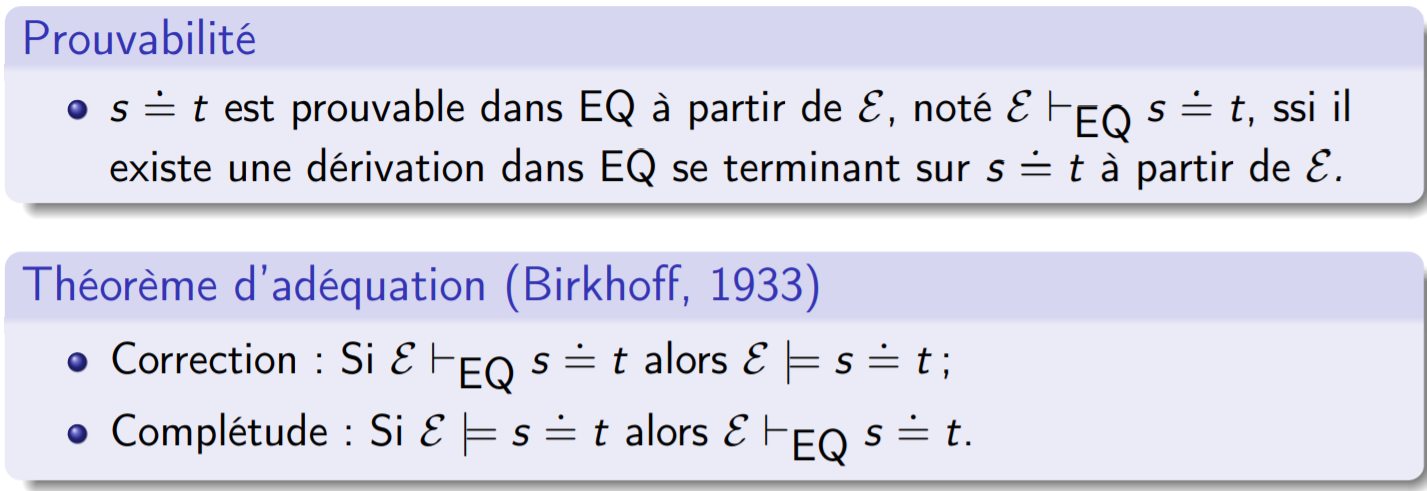


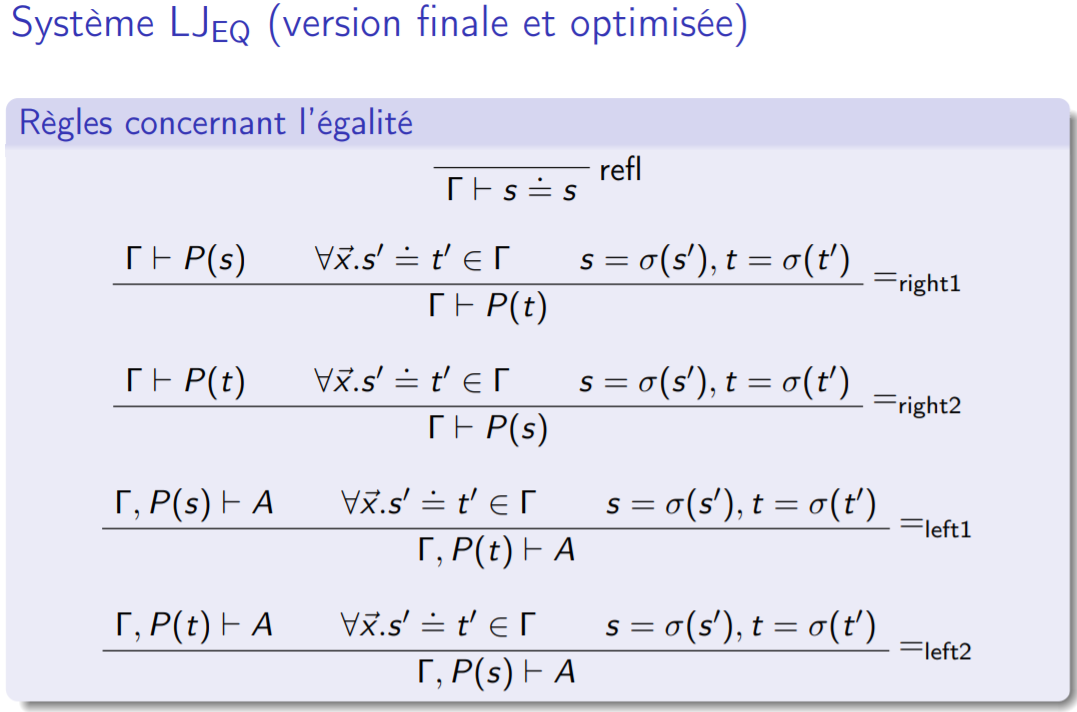


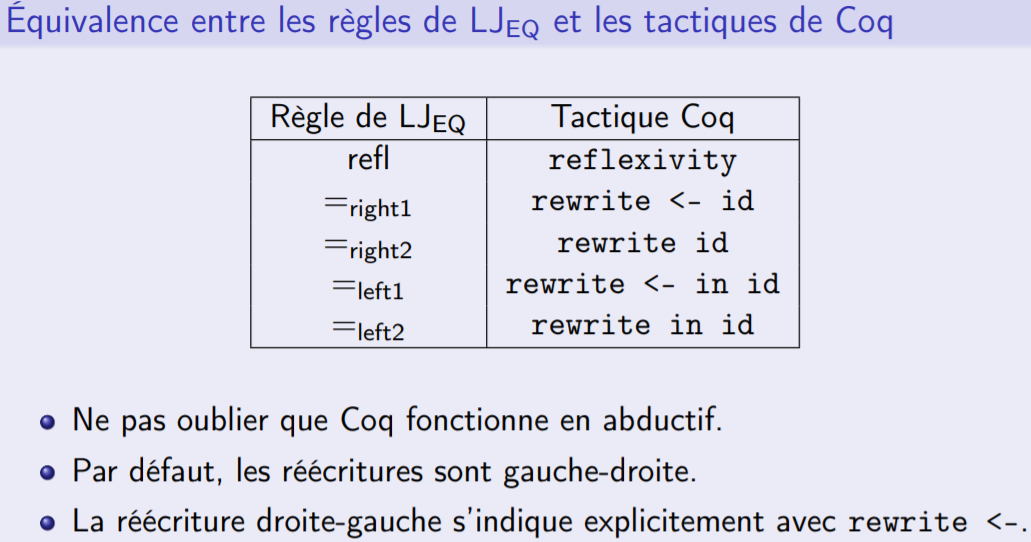
Position substitution :











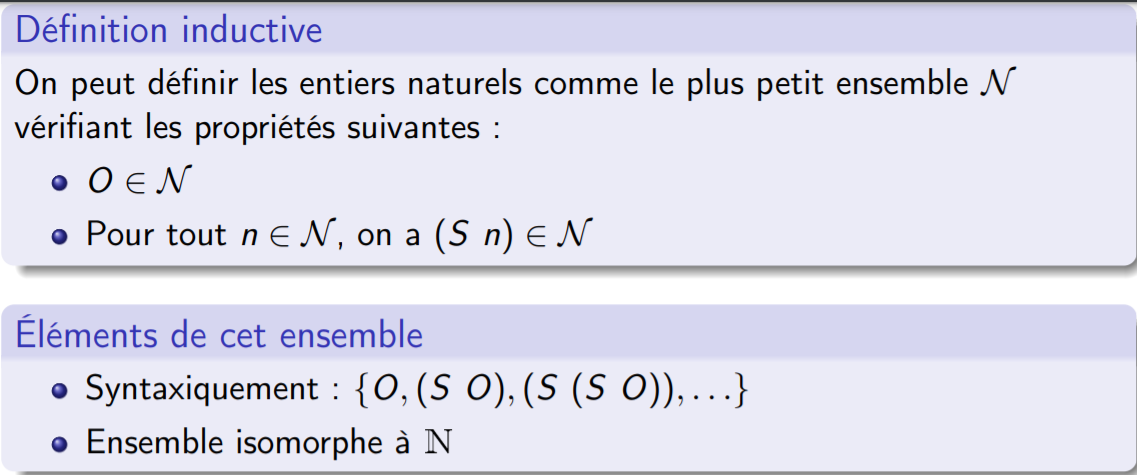
La plupart du temps, on commence souvent par un *trans*.

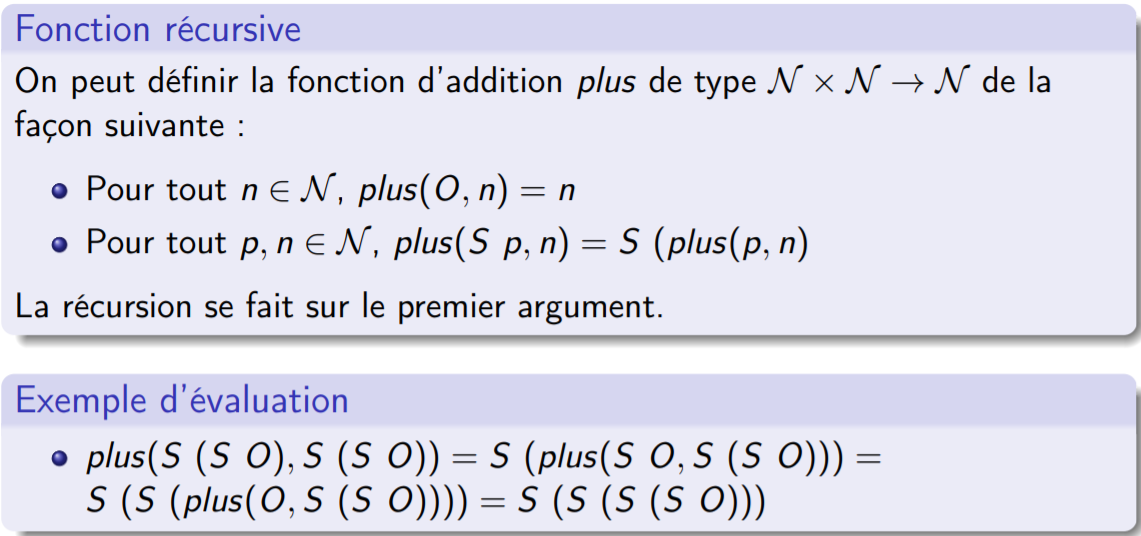
### **Types inductifs :**

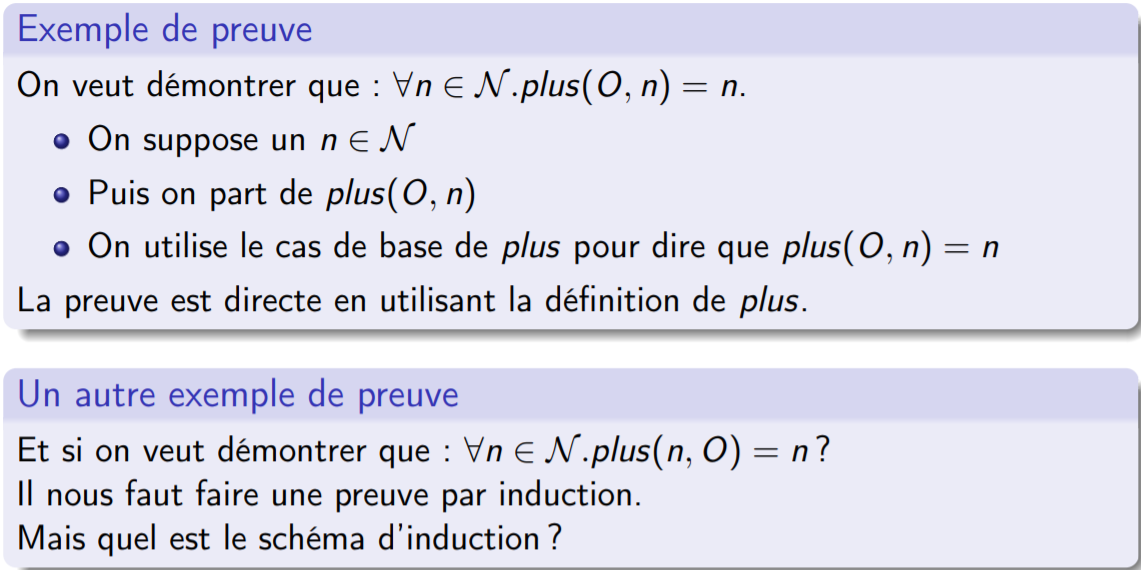
On peut tout formaliser à base de types inductifs ;

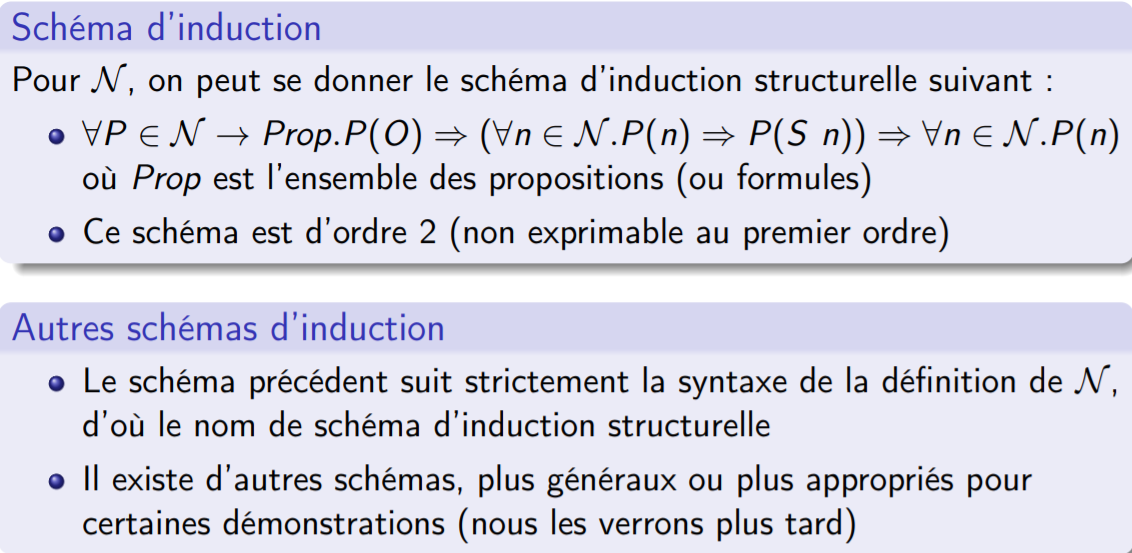
Types inductifs pour les types de données ;

Le premier exemple, les entiers naturels :

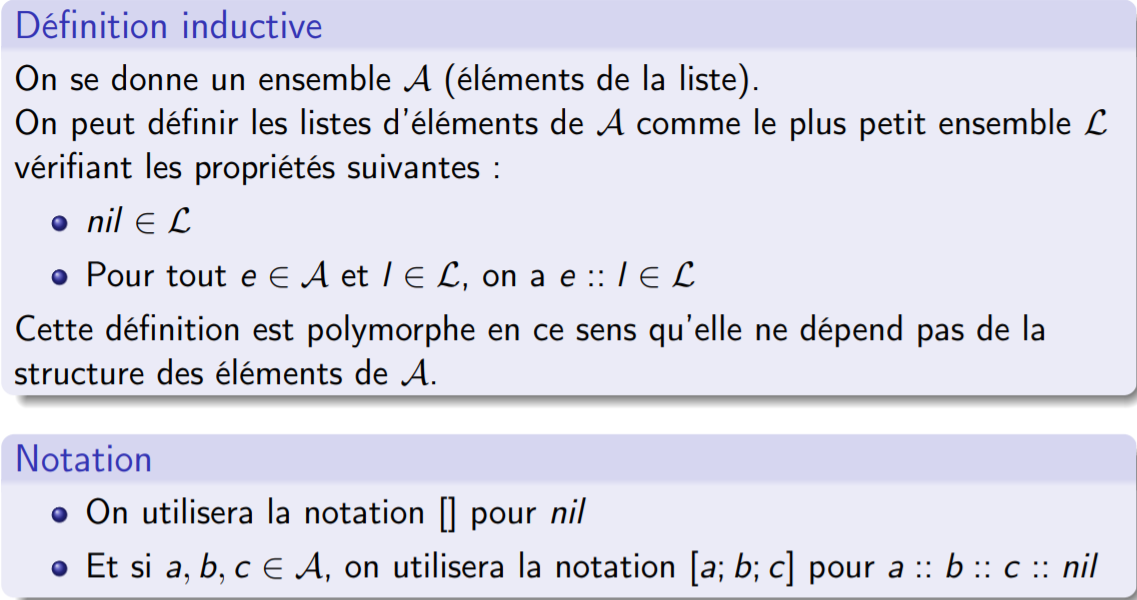


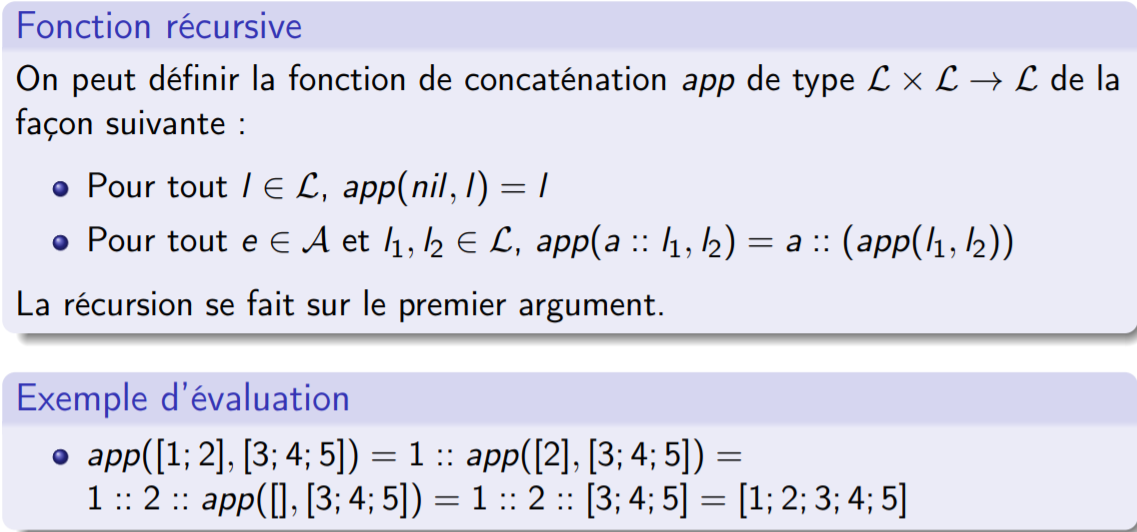


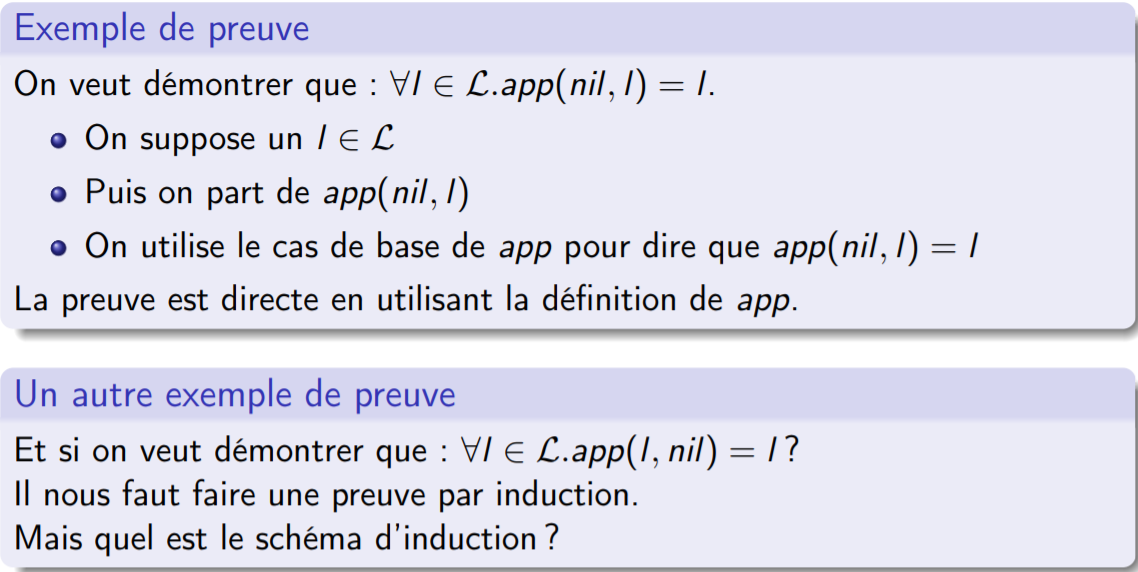


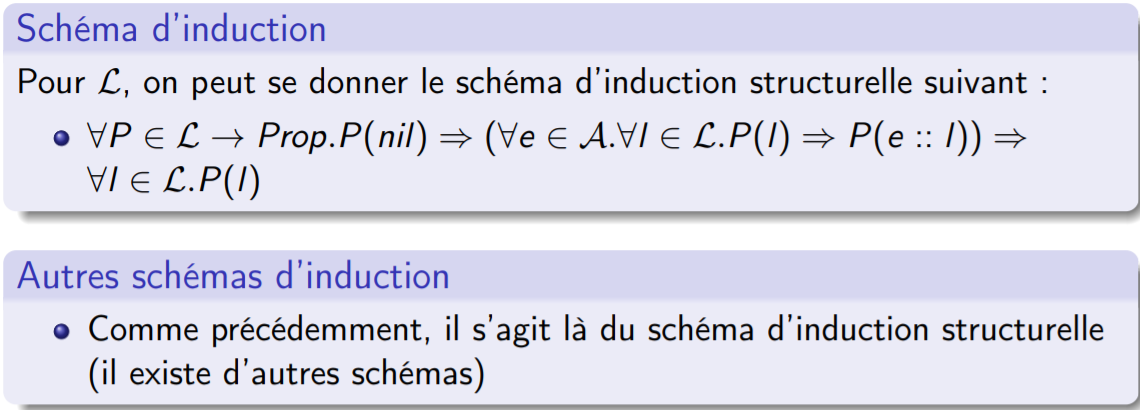


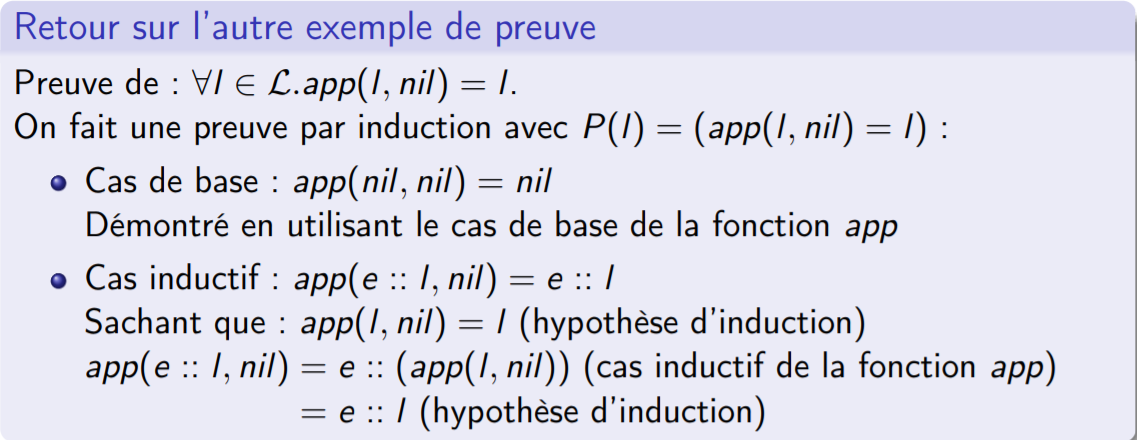
**Un deuxième exemple avec les listes :**





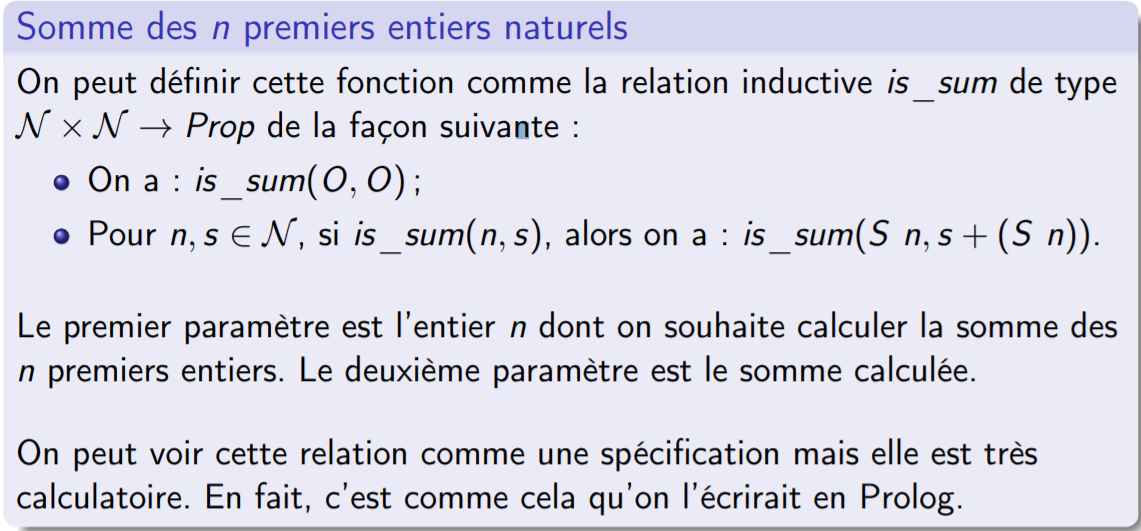


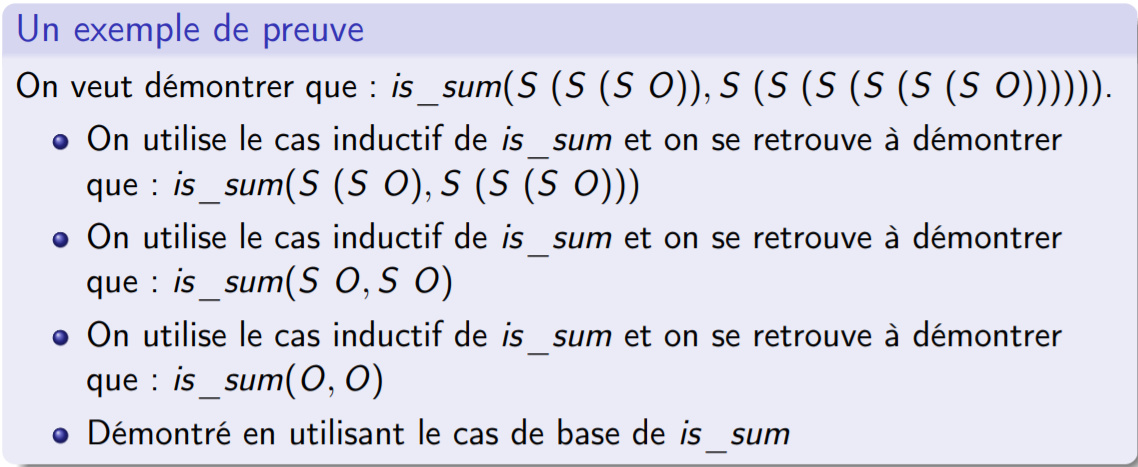




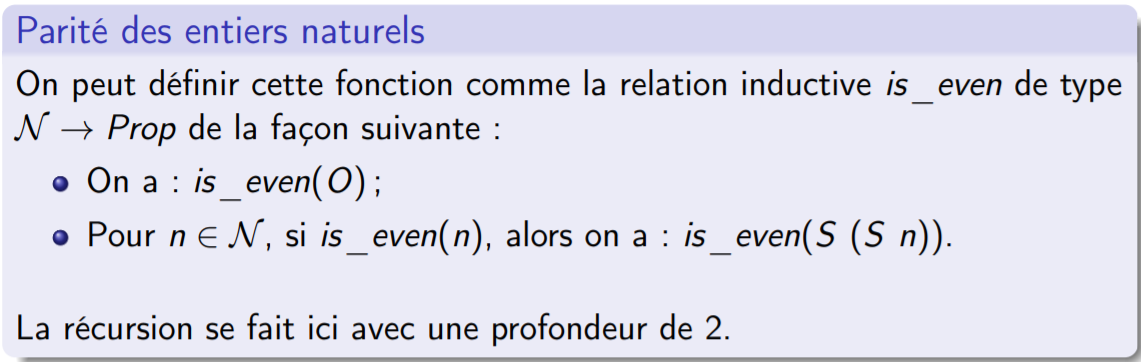
**Relations inductives :**

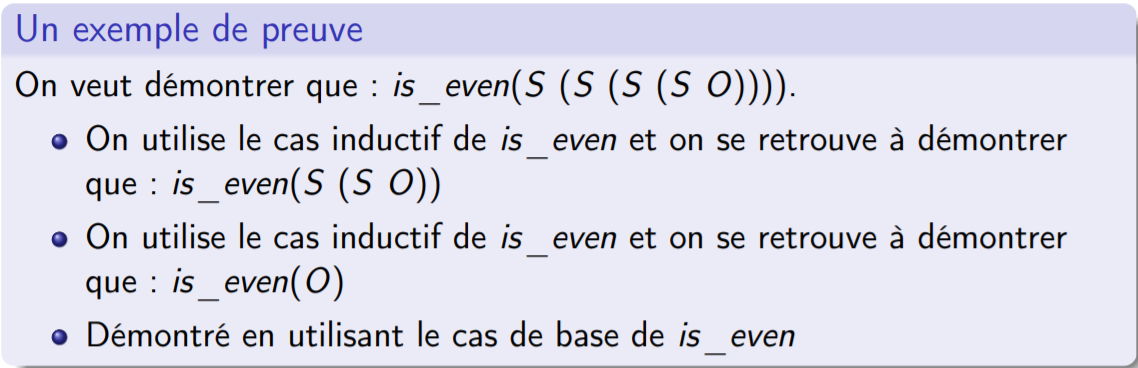
**Exemple avec la somme des n entiers naturels :**



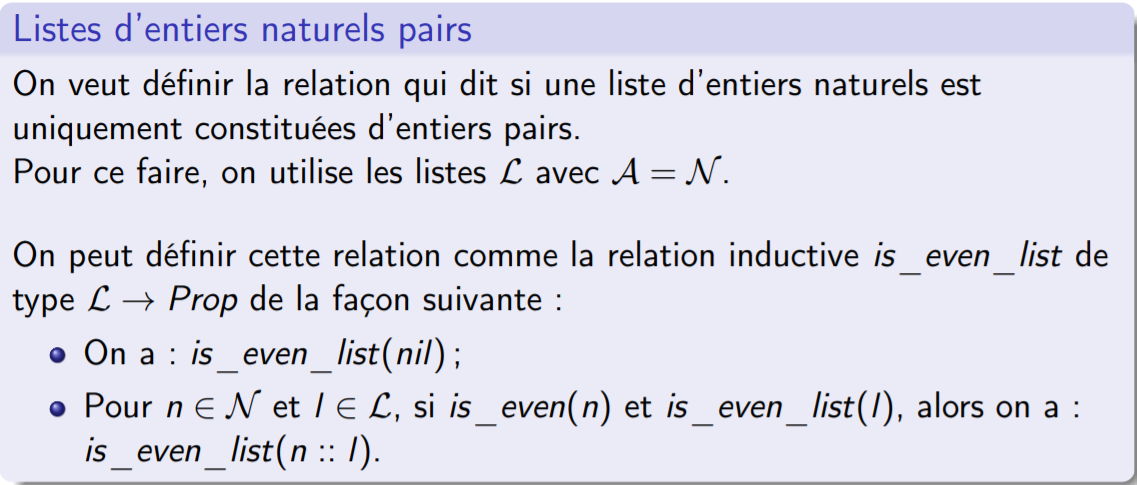


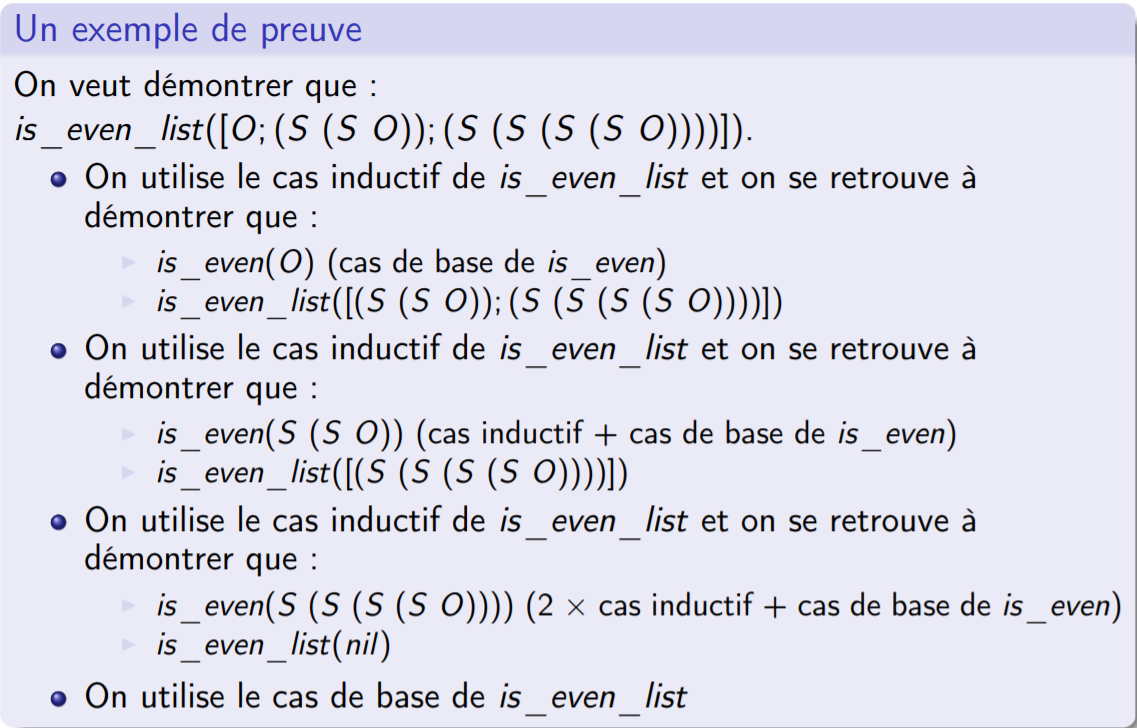
Avec la parité des entiers naturels :





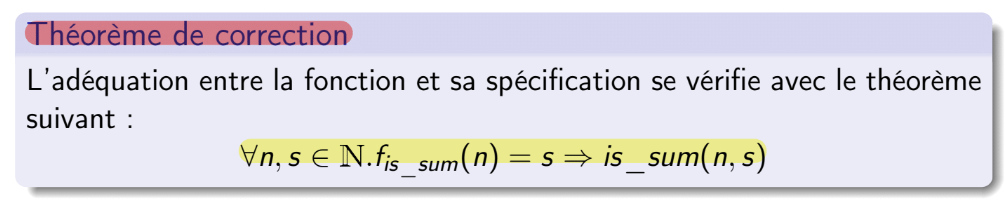
Dernier exemple : Listes d’entiers naturels pairs :



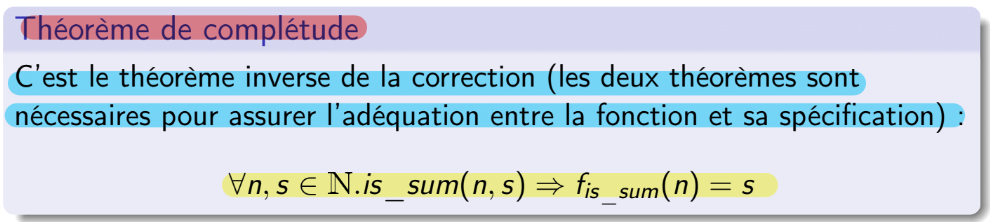


Regarder dans le cours pour les listes.

**Théorème de correction :**



**Théorème de complétude :**



<https://docs.google.com/document/d/16aqRXxSCsyvrnAtyCMCUDMQrhUFysrTpXXe8TItOnZo/edit?usp=sharing>

<https://www.lama.univ-savoie.fr/pagesmembres/hirschowitz/teaching/logique-et-lambda-calcul/>

<https://www.lri.fr/~paulin/MPRI/notes/cours007.html>

<https://perso.liris.cnrs.fr/xavier.urbain/ens/LIFLC/CoqNat.pdf>

<https://learntutorials.net/fr/coq/topic/8335/utiliser-la-tactique>

<https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html>

### **Preuves par induction :**

On définit la relation inductive is\_sum de type N × N → Prop de la façon suivante :

1. On a : is\_sum(0, 0);
2. Pour n,s ∈ N, si is\_sum(n,s), alors on a : is\_sum(S(n),s + S(n))

**Induction structurelle :**

| **Théorème de correction** L’adéquation entre la fonction et sa spécification se vérifie avec le théorème suivant : ∀n,s ∈ N.fis\_sum(n) = s ⇒ is\_sum(n,s)  La preuve se fait par induction sur n. On utilise le schéma d’induction structurelle sur N :  ∀P ∈ N → Prop.P(0) ⇒ (∀n ∈ N.P(n) ⇒ P(S(n))) ⇒ ∀n ∈ N.P(n)  Dans notre cas : P(n) = ∀s ∈ N.fis\_sum(n) = s ⇒ is\_sum(n,s)  **Preuve :** On applique le schéma d’induction et on doit démontrer :   1. Cas de base : ∀s ∈ N.fis\_sum(0) = s ⇒ is\_sum(0,s)   On calcule fis\_sum(0), ce qui donne : ∀s ∈ N.0 = s ⇒ is\_sum(0,s)  On remplace s par 0, et on doit démontrer is\_sum(0, 0), qui est le cas de base de la spécification inductive de la relation is\_sum.   1. Cas inductif : pour n ∈ N, ∀s ∈ N.fis\_sum(S(n)) = s ⇒ is\_sum(S(n),s)   sous l’hypothèse d’induction : ∀s ∈ N.fis\_sum(n) = s ⇒ is\_sum(n,s)  On calcule fis\_sum(S(n)), ce qui donne : ∀s ∈ N.fis\_sum(n) + S(n) = s ⇒ is\_sum(S(n),s)  On remplace s par fis\_sum(n) + S(n), et on doit démontrer : is\_sum(S(n), fis\_sum(n) + S(n))  On applique le cas inductif de la spécification de is\_sum, et on doit démontrer :  is\_sum(n, fis\_sum(n)). On applique l’hypothèse d’induction avec s = fis\_sum(n), et il nous  reste à démontrer que fis\_sum(n) = fis\_sum(n), ce qui est trivial. |
| --- |

| Théorème de complétude C’est le théorème inverse de la correction (les deux théorèmes sont nécessaires pour assurer l’adéquation entre la fonction et sa spécification) :  ∀n,s ∈ N.is\_sum(n,s) ⇒ fis\_sum(n) = s  **Preuve** : La preuve se fait par induction sur la relation is\_sum. On utilise le schéma d’induction lié à la relation is\_sum (ce schéma peut toujours être qualifié de structurel ; il suit la définition de la relation) : ∀P ∈ N × N → Prop.  P(0, 0) ⇒ (∀n,s ∈ N.is\_sum(n,s) ⇒ P(n,s) ⇒ P(S(n),s + S(n))) ⇒ ∀n,s ∈ N.is\_sum(n,s) ⇒ P(n,s). On applique le schéma d’induction avec P(n,s) = (fis\_sum(n) = s). On doit démontrer  Cas de base : fis\_sum(0) = 0  On calcule fis\_sum(0), ce qui donne 0 = 0 (trivial).  On applique le schéma d’induction avec P(n,s) = (fis\_sum(n) = s).  On doit démontrer :  Cas inductif : pour n,s ∈ N, fis\_sum(S(n)) = s + S(n)  sous les hypothèses d’induction : is\_sum(n,s) et fis\_sum(n) = s  On calcule fis\_sum(S(n)), ce qui donne : fis\_sum(n) + S(n) = s + S(n)  On applique le schéma d’induction avec P(n,s) = (fis\_sum(n) = s). On doit démontrer :  On remplace s par fis\_sum(n), ce qui donne (trivial) : fis\_sum(n) + S(n) = fis\_sum(n) + S(n) |
| --- |

**Remarque** : La relation is\_sum est définie.

is\_sum\_O et is\_sum\_S sont ses constructeurs.

Ils doivent être considérés comme des axiomes.

Le schéma d’induction structurelle is\_sum\_ind est généré.

| **Induction fonctionnelle :**  On définit la relation inductive is\_even de type N → Prop de la façon suivante :   1. On a : is\_even(0); 2. Pour n ∈ N, si is\_even(n), alors on a : is\_even(S(S(n))).   **Théorème de correction :** L’adéquation entre la fonction et sa spécification se vérifie avec le théorème suivant : ∀n ∈ N.fis\_even(n) = > ⇒ is\_even(n)  **Par induction structurelle sur n :**   1. Cas de base : fis\_even(0) = > ⇒ is\_even(0). On applique simplement le cas de base de is\_even. 2. Cas inductif : pour n ∈ N, fis\_even(S(n)) = > ⇒ is\_even(S(n)) sous l’hypothèse d’induction : fis\_even(n) = > ⇒ is\_even(n)   On doit refaire une deuxième induction sur n :  **Cas inductif :** Cas de base : fis\_even(1) = > ⇒ is\_even(1)  On calcule fis\_even(1), ce qui donne : ⊥ = > ⇒ is\_even(1)  ce qui est trivial car ⊥ = > est faux.  On doit refaire une deuxième induction sur n :  2) Cas inductif : fis\_even(S(S(n))) = > ⇒ is\_even(S(S(n)))  sous les hypothèses : fis\_even(S(n)) = > ⇒ is\_even(S(n))  (fis\_even(n) = > ⇒ is\_even(n)) ⇒ fis\_even(S(n)) = > ⇒ is\_even(S(n))  On calcule fis\_even(S(S(n))) et on applique le cas inductif de la relation  is\_even, et on doit démontrer is\_even(n) sous les hypothèses :  fis\_even(n) = > fis\_even(S(n)) = > ⇒ is\_even(S(n))  (fis\_even(n) = > ⇒ is\_even(n)) ⇒ fis\_even(S(n)) = > ⇒ is\_even(S(n)) |
| --- |

**Spécifications :**

* Les spécifications sont parfois plus abstraites ;
* Elles ne contiennent pas forcément un algorithme ;
* C’est même mieux si elles n’en contiennent pas ;
* Si elles contiennent un algorithme, elles n’en imposent pas forcément
* un au niveau de l’implantation (il faudra en démontrer l’équivalence).

**Relation bien fondée :**

Soit une relation binaire R sur un ensemble A, c’est-à-dire que R ⊆ A × A.

La relation R sera bien fondée dans A s’il n’existe pas de chaînes descendantes infinies, c’est-à-dire de suite (ui) dans A telle que ui+1 R ui pour tout i.

Une fonction f sur A sera définie par induction bien fondée si elle est de la

forme suivante : f (x) = g(x, f|inf(x))

**Prouver = Programmer :**

**Des principes de preuve**

* Raisonnement par l’absurde : une façon de prouver qu’un objet existe est de supposer qu’il n’existe pas, et d’aboutir à une contradiction. Alors, puisqu’il ne peut pas ne pas exister, c’est donc qu’il existe. Mais on n’est pas plus avancé s’il s’agit de le trouver !
* Règle d’élimination des doubles négations. Pourtant si naturelle ! Le tiers exclu est souvent modélisé par cette règle.

**Des théorèmes**

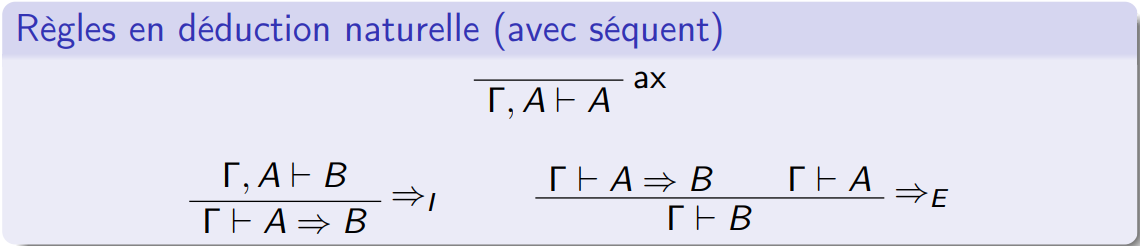
* Avec un axiome en moins, on prouve moins de choses.
* Par exemple, le théorème qui affirme que toute suite croissante majorée converge n’est pas valide en mathématiques constructives.
* Les mathématiques sont à réinventer !

**Par induction sur les formules**

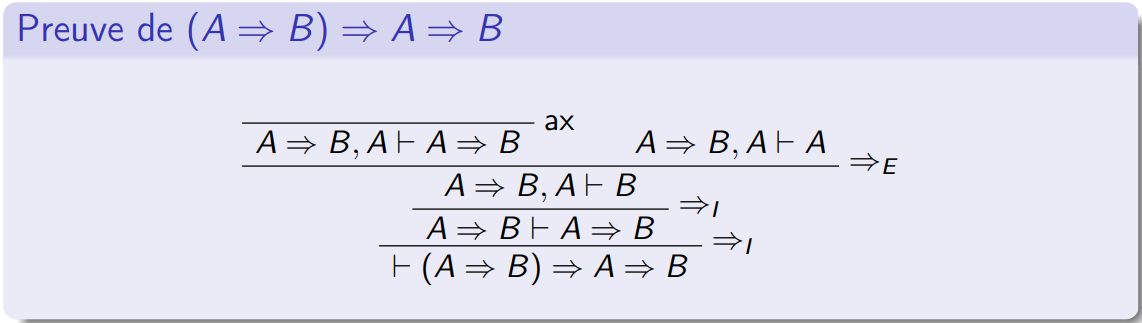
* Une preuve de A ⇒ B est une fonction qui associe à une preuve de A une preuve de B
* Une preuve de A ∧ B est un couple (π1, π2), où π1 est une preuve de A et π2 une preuve de B
* Une preuve de A ∨ B est soit une preuve de A, soit une preuve de B
* Une preuve de ∀x.A(x) est une fonction qui associe à tout objet t une preuve de A(t)
* Une preuve de ∃x.A(x) est un couple (t, π), où t est un objet et π est une preuve de A(t)
* Une preuve de ¬A (vue comme A ⇒ ⊥) est une fonction qui associe à toute preuve de A une preuve de ⊥
* On désigne par I la preuve de >, et il n’existe pas de preuve ⊥

**Curry** : analogie entre les preuves dans les systèmes à la Hilbert et la logique combinatoire

**Howard** : analogie entre les preuves en déduction naturelle intuitionniste et les termes du λ-calcul typé



Exemple :



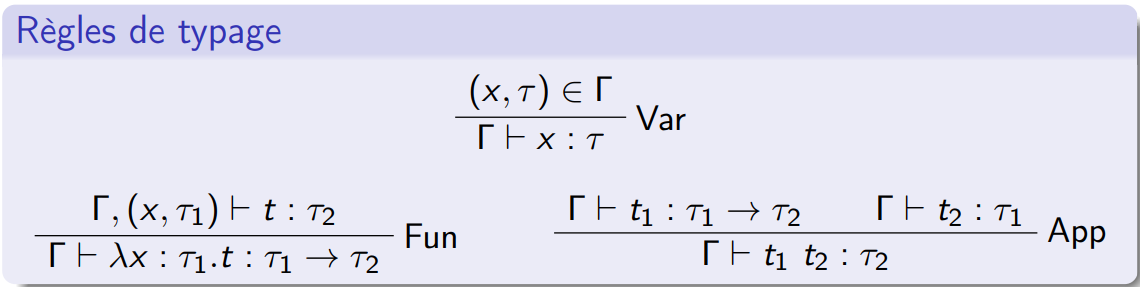
**λ-calcul simplement typé**

**Termes** :

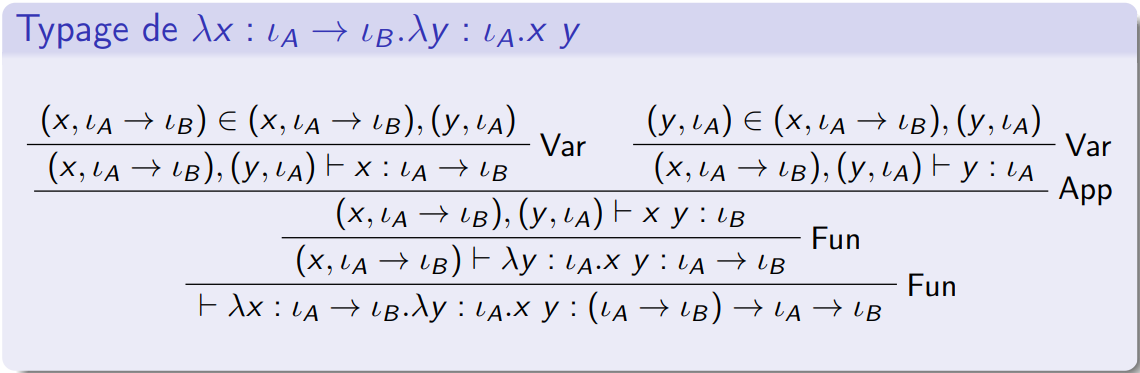
* Les variables x, y, . . . sont des variables
* Si x est une variable, τ un type, et t un terme, alors λx : τ.t est un terme (notation à la Church)
* Si t1 et t2 sont des termes, alors t1 t2 est un terme

**Types** :

* Les types de base ι1, ι2, . . . sont des types
* Si τ1 et τ2 sont des types, alors τ1 → τ2 est un type



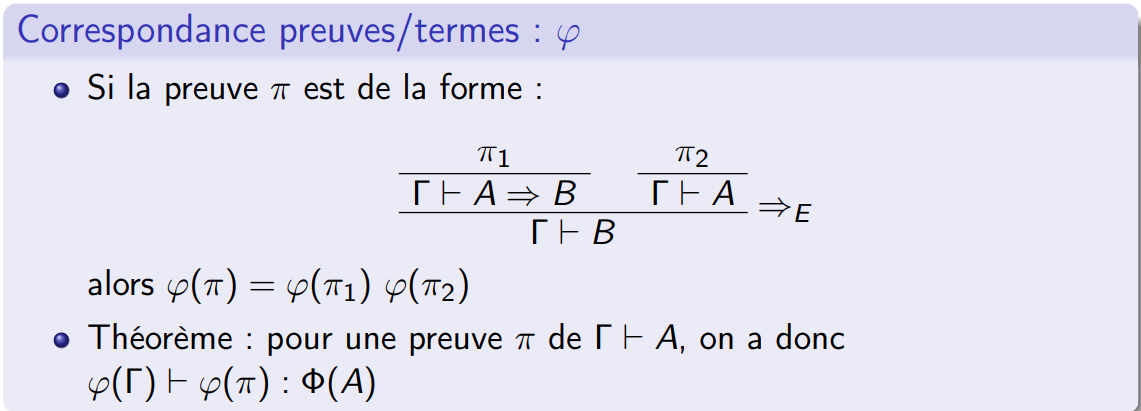
**Exemple :**



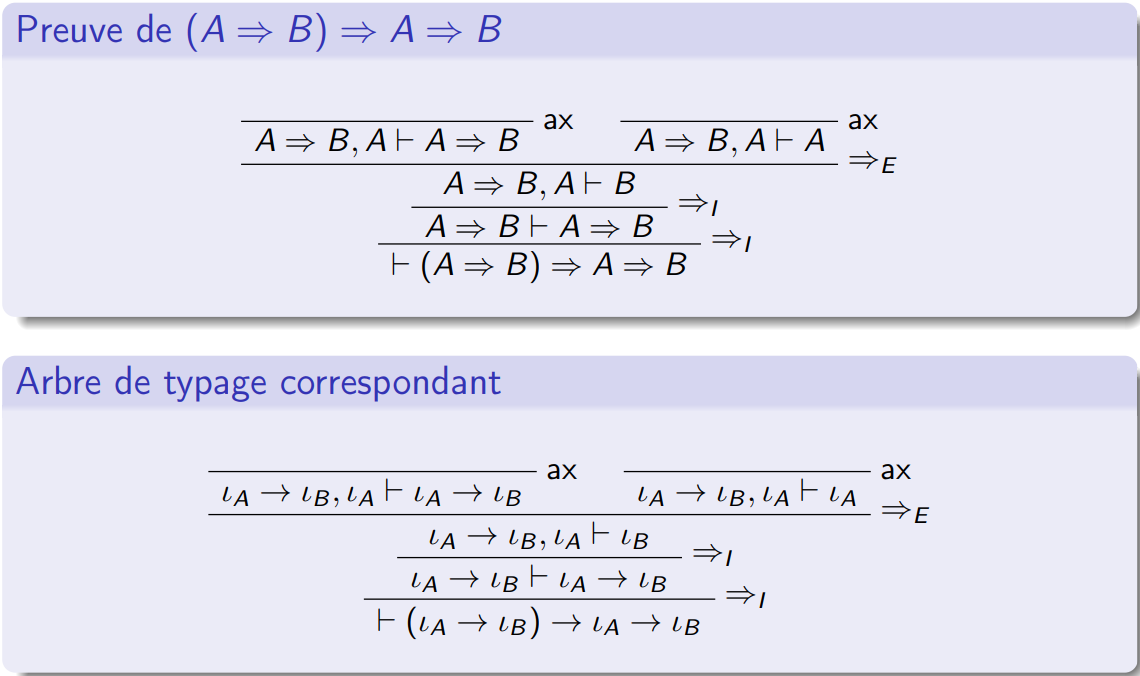
**Isomorphisme de Howard :**

Correspondance formules/types : Φ

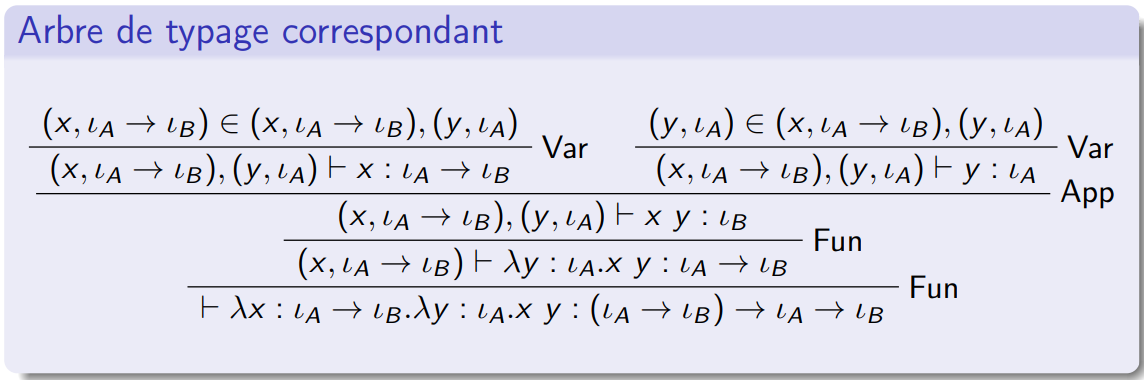
* Φ(A) = ιA ;
* Φ(A ⇒ B) = Φ(A) → Φ(B).



Exemple :



Donne : (voir cours prouver = programmer diapo 49)

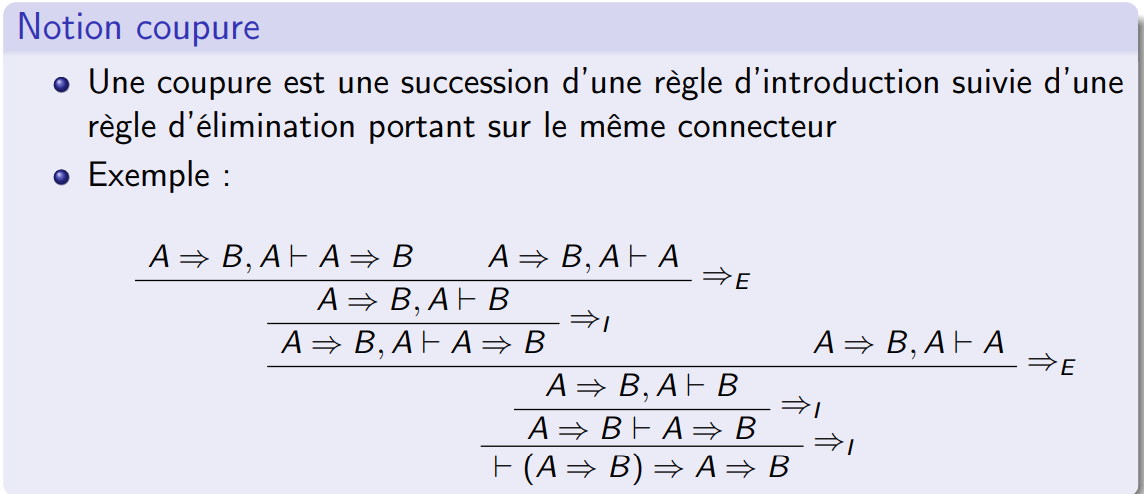


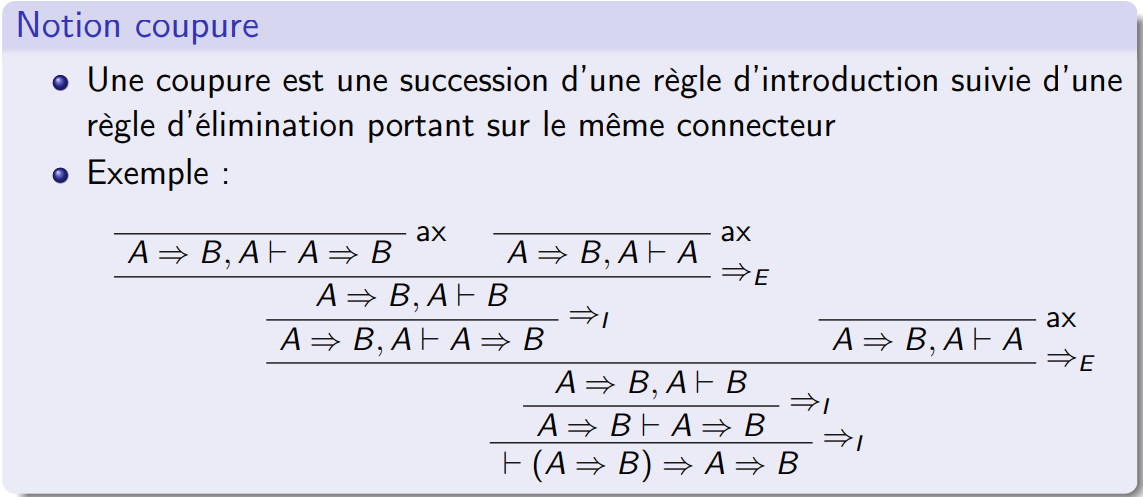
**Logique implicative minimale**

Démontrer les propositions suivantes en déduction naturelle et en extraire les termes correspondants en λ-calcul simplement typé :

A ⇒ B ⇒ A : λx : τA.λy : τB .x

(A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C : λf : τA → τB → τC .λg : τA → τB .λx : τA.f x (g x)





**Théorème d’élimination des coupures**

* « Hauptsatz » de Gentzen (1934)
* Toute formule qui possède une preuve faisant usage d’une coupure,
* possède aussi une preuve sans coupure
* Il existe un algorithme qui prend une preuve d’une formule et la transforme en une preuve sans coupure de la même formule
* Cela permet, entre autres, de montrer la cohérence de la logique

|  |
| --- |
|  |

Comparaison des termes preuves :

t1 = λx : τA → τB .λy : τA.(λz : τA.x z) y

t2 = λx : ιA → ιB .λy : ιA.x y

On observe que : t1 →β t2

* L’élimination des coupures correspond donc à la β-réduction, c’est-à-dire à l’exécution des fonctions et donc des programmes !
* Une coupure correspond ainsi à un β-rédex
* Comme l’élimination des coupures est un processus qui termine, cela signifie que tous les programmes sont terminants dans ce calcul

∧ : produit cartésien, couple/projections ;

∨ : union disjointe, injections/filtrage ;

¬ : revient à l’implication (¬A ≡ A ⇒ ⊥) ;

∀ : produit (implication dépendante), fonction/application ;

∃ : sigma (produit cartésien dépendant), couple/projections.

### **Preuves de programmes impératifs :**

Définition rigoureuse de l’exécution

* Tout comportement est spécifié (même les cas d’erreurs) ;
* Plus d’ambiguïtés pour l’utilisateur (ordre d’évaluation).

Démonstration formelle de propriétés

* Équivalences sémantiques (si différentes sémantiques) ;
* Équivalences de programmes (syntaxiquement différents) ;
* Correction de transformations de programmes ;
* Propriétés relatives au typage :

- Correction du typage vis-à-vis de la sémantique ;

- Préservation du typage par la sémantique.

**Syntaxe**

* Définition préalable de la syntaxe (abstraite) du langage ;
* Utilisation d’une structure arborescente (AST).

**Différentes sémantiques**

* Sémantique opérationnelle naturelle (à grands pas) ;
* Sémantique opérationnelle structurée (à petits pas) ;
* Sémantique dénotationnelle (théorie des domaines) ;
* Sémantique axiomatique (logique de Hoare).

**Sémantique opérationnelle naturelle (à grands pas) :**

Expressions arithmétiques : e ::= n | e1 + e2 | e1 − e2 | e1 × e2 | e1/e2 où n ∈ Z

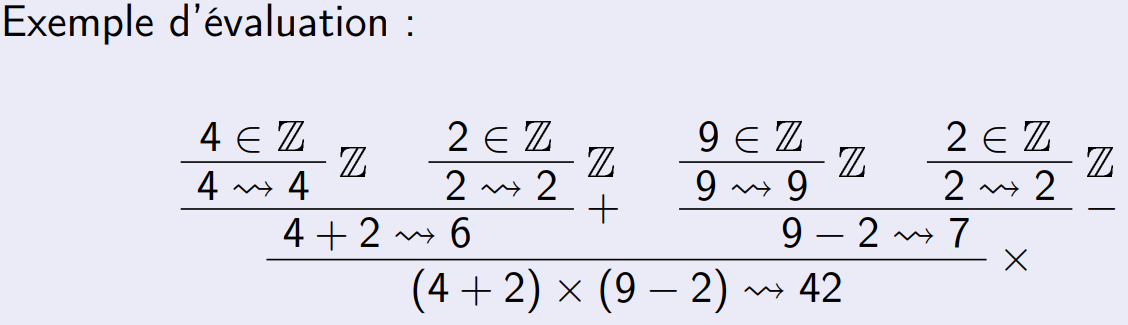
Valeurs : ve ::= n, où n ∈ ℤ ;

Sémantique : relation « e ↝ ve » ;

Règles en langage naturel :

* Si n ∈ ℤ, alors n n ; ↝
* Si e1 ↝ v1 et e2 ↝ v2, alors e1 + e2 ↝ v1 +ℤ v2 ;
* Si e1 ↝ v1 et e2 ↝ v2, alors e1 − e2 ↝ v1 −ℤ v2 ;
* Si e1 ↝ v1 et e2 ↝ v2, alors e1 × e2 ↝ v1 ×ℤ v2 ;
* Si e1 ↝ v1 et e2 ↝ v2, alors e1/e2 ↝ v1/ ℤv2.





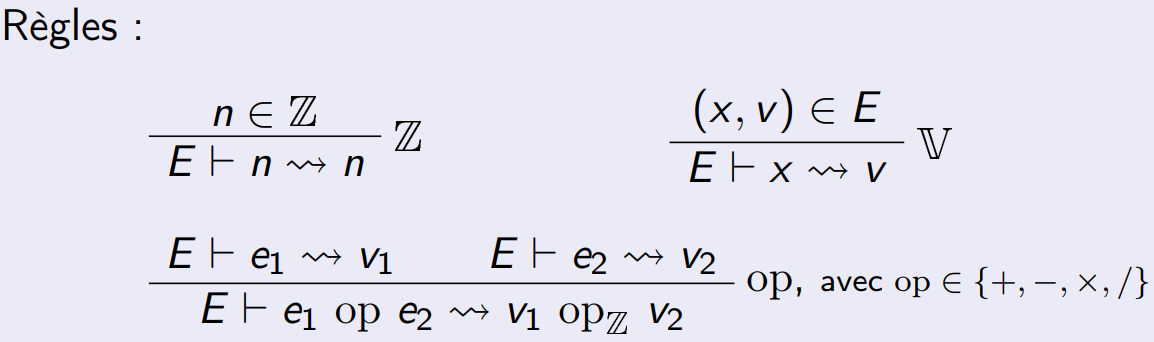
Expressions arithmétiques avec variables

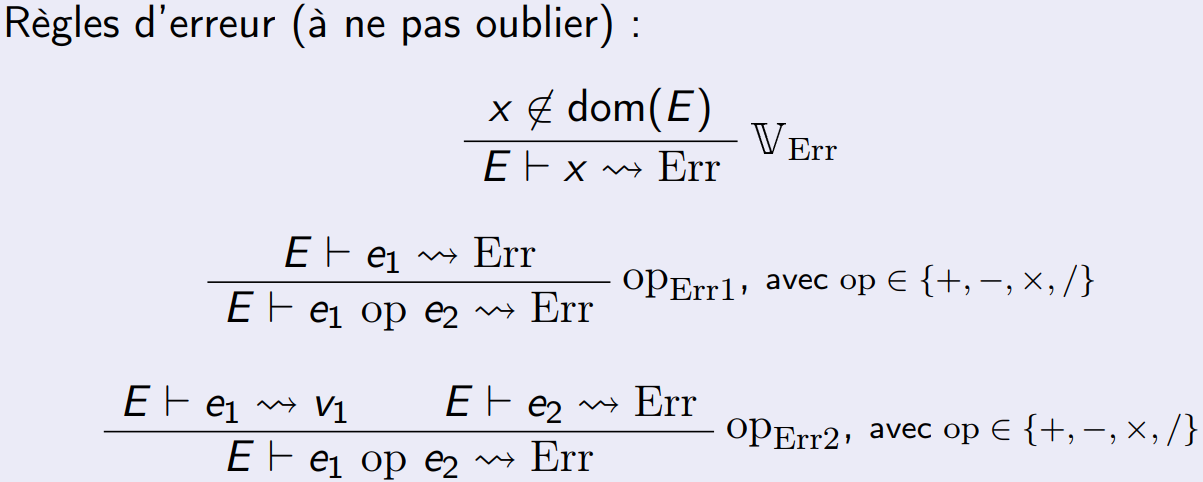
e ::= n | x | e1 + e2 | e1 − e2 | e1 × e2 | e1/e2 où n ∈ Z et x ∈ V (ensemble de noms de variables).

Valeurs : ve ::= n | Err, où n ∈ Z ;

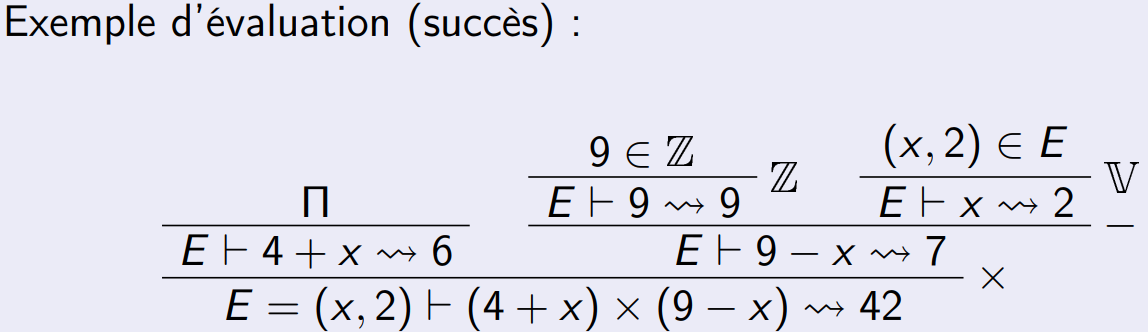
Contextes d’exécution : E = (x1, v1),(x2, v2), . . . ,(xn, vn);

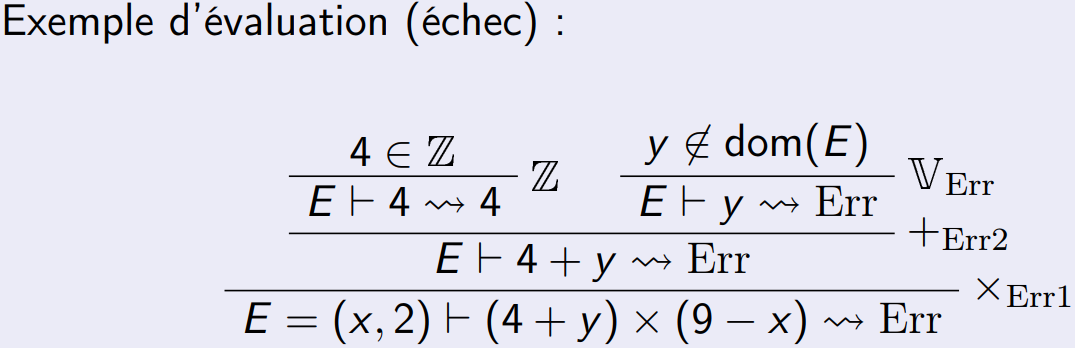
Sémantique : relation « E |- e ↝ ve » ;





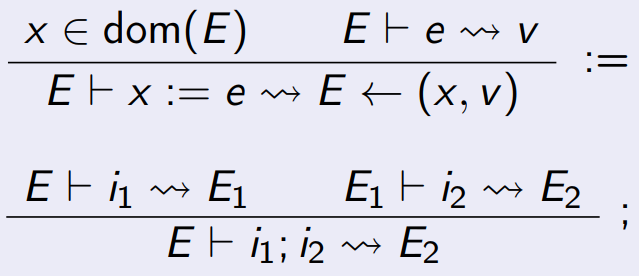
<https://www.youtube.com/watch?v=kjxdelbo9C4&list=PLA72M-qSGPm2bZlhxYB-ePerW0U8nPn4H>





Règles

Instruction : affectation et séquence : i ::= x := e | i1; i2 où x ∈ V (ensemble de noms de variables).

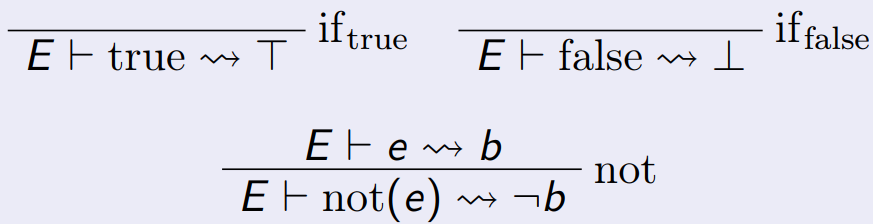


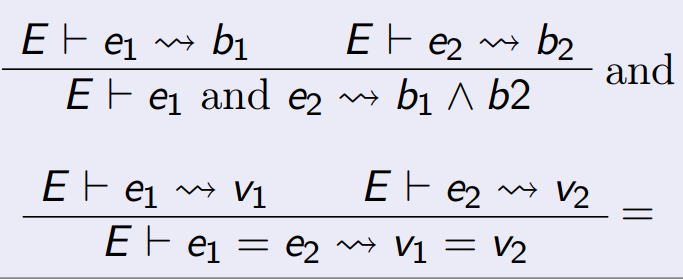
Instruction : Conditionnelle :

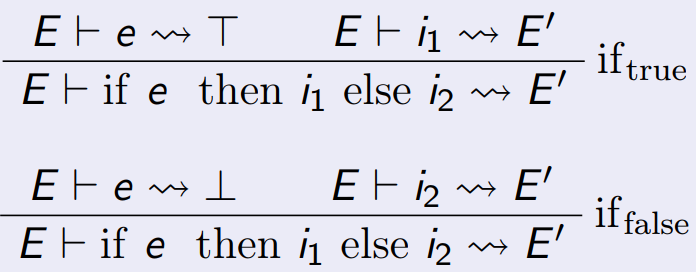
e ::= . . . | true | false | not(e) | e1 and e2 | e1 or e2

| e1 = e2 | e1! = e2 | e1 < e2 | e1 ≤ e2 | e1 ≥ e2 | e1 > e2 ;

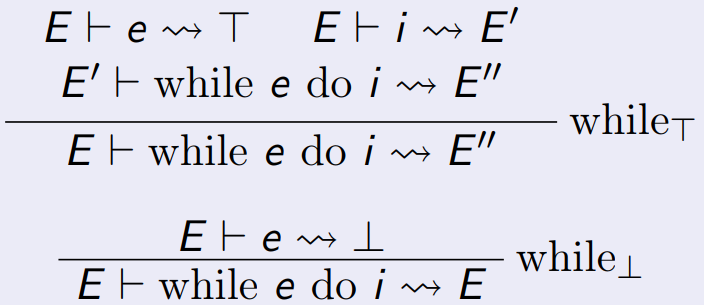
i ::= . . . | if e then i1 else i2.

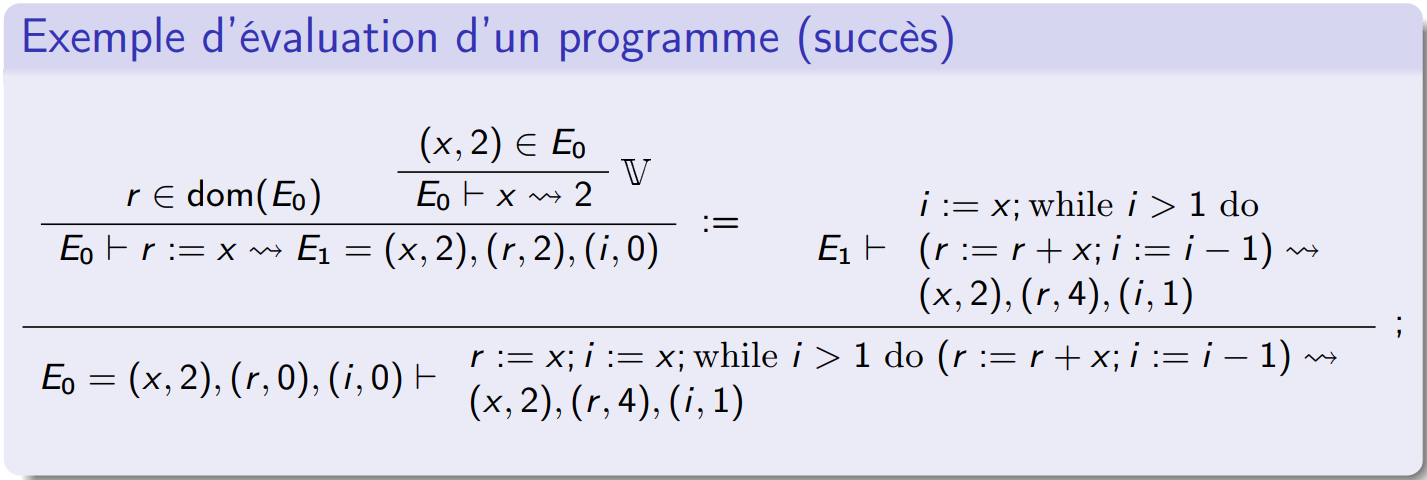






i ::= . . . | while e do i.



****

**Le langage résumé :**

Petit noyau impératif

* Expressions entières et booléennes ;
* Instructions d’affectation, de conditionnelle, et de boucle.

**Expressions et instructions :**

e ::= n | x | e1 + e2 | e1 − e2 | e1 × e2 | e1/e2

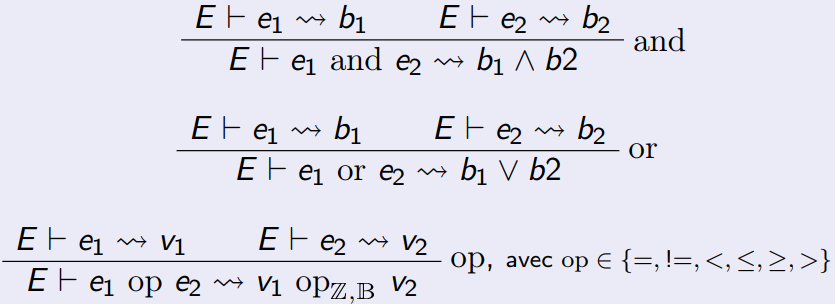
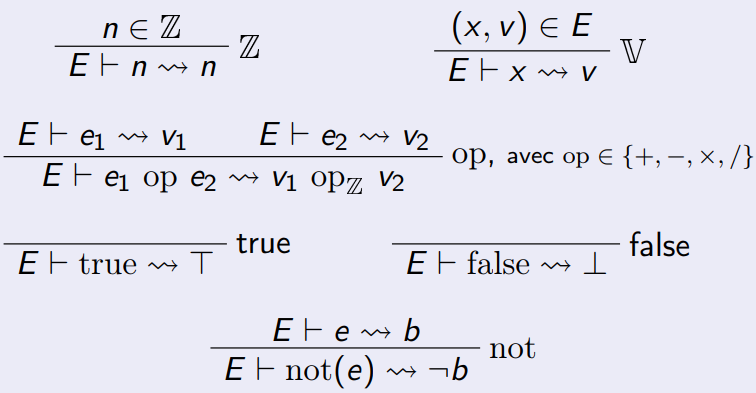
| true | false | not(e) | e and e | e or e

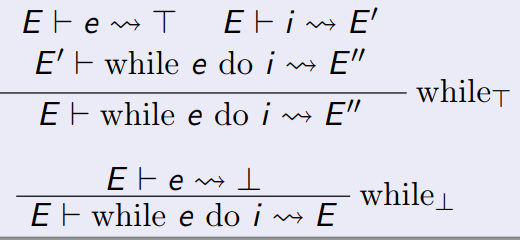
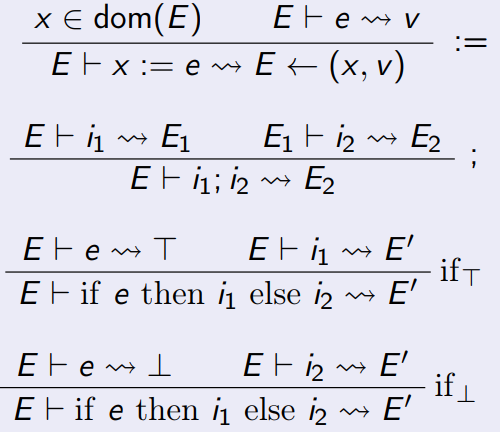
| e = e | e != e | e < e | e ≤ e | e ≥ e | e > e

où n ∈ Z et x ∈ V (ensemble de noms de variables) ;

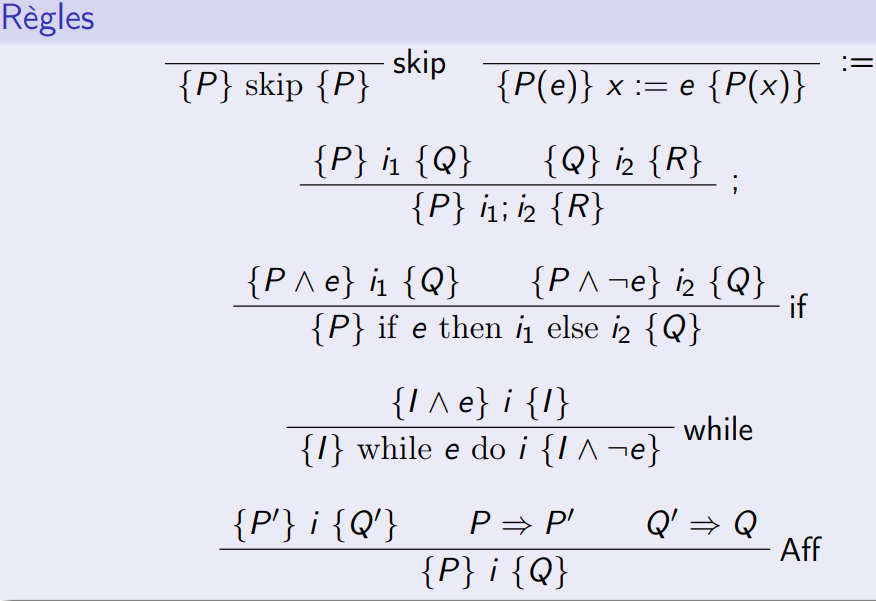
i ::= skip | x := e | i; i | if e then i else i | while e do i.

**Règle** :





**Logique de Hoare :**

* Triplet noté : {P} i {Q}, où P et Q sont des assertions logiques, et i une instruction ;
* Assertions logiques : exprimées en logique du premier ordre, où les atomes sont les expressions de notre langage ;
* Un triplet de Hoare {P} i {Q} est valide si pour tous états E1 et E2 tels que si P est vraie dans E1 et E1 ` i E2 (i termine), alors Q est vraie dans E2.

Exemples de triplets de Hoare valides :

{x = 1} x := x + 2 {x = 3} ;

{x = y} x := x + y {x = 2 × y}.

Exemple :

