Références

- documentation général :
  - Coq'Art (Y.Bertot and P.Castéran)
  - Certified Programming with Dependent Types(A.Chlipala)
- points techniques spécifiques :
  - type classes : On typeclasses(B.Pierce)
- preuves détaillées, domaine spécifiques
  - On language semantics and proofs of programs(B.Pierce)
- survival Kit(D.Pichardie, S.Blazy)

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lpf

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom $\times$ définition
- définition
  - types, valeurs, fonctions $\in$ lpf
  - + type dépendants

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lpf
  - + type dépendants
    - données dépendantes
    - propriétés + preuves

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lpf
  - + type dépendants
    - données dépendantes $\in$ lpf du futur ?
    - propriétés + preuves

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lpf
  - + type dépendants
    - données dépendantes $\in$ lpf du futur ?
    - propriétés + preuves $\notin$ lpf

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lpf `Type`
  - + type dépendants
    - données dépendantes $\in$ lpf du futur ? `Type`
    - propriétés + preuves $\notin$ lpf `Prop`

Qu'est-ce-que Coq ?

Principalement un langage fonctionnel pur (lfp) à la Haskell
+ un supplément

- programme = séquence de déclarations
- déclaration = nom × définition
- définition
  - types, valeurs, fonctions $\in$ lfp `Type`
  - + type dépendants
    - données dépendantes $\in$ lfp du futur ? `Type`
    - propriétés + preuves $\notin$ lfp `Prop`

`Type` VS `Prop`

- types des types : `Set`, `Type` et `Prop`
- `0 : nat : Set` `: Type`^{1}
- `le_n0 : 0 <= 0 : Prop`
- en fait (pour éviter les paradoxes) :

  `Type`^{0} = `Set` / `Prop` } : `Type`^{1} : `Type`^{2} ...∞

- une définition donnée est :
  - une donnée si dans `Type`
  - une propriété si dans `Prop`

Type VS Prop

```
3                   :        nat                  : Type
Nat.add             :        nat -> nat -> nat    : Type
1 ?= 3              :        bool                 : Type
```

```
eq_refl 0           :        0 = 0                : Prop.
(O_S 1)             :        0 <> 2               : Prop.
Nat.le_0_1          :        0 <= 1               : Prop.
fun n:nat => eq_refl n:    forall (n:nat), n = n  : Prop.
```

- types contiennent des termes (types dépendants),
- typiquement les propriétés mais pas seulement
- Pretty printing `->` = →, `forall` = ∀

Type VS Prop

termes | types | types (sortes)

```
        3             :          nat                 : Type

     Nat.add          :    nat -> nat -> nat         : Type

     1 ?= 3           :         bool                 : Type
```

```
    eq_refl 0         :          0 = 0               : Prop.

     (O_S 1)          :          0 <> 2              : Prop.

    Nat.le_0_1        :          0 <= 1              : Prop.

fun n:nat => eq_refl n:   forall (n:nat), n = n      : Prop.
```

- types contiennent des termes (types dépendants),
- typiquement les propriétés mais pas seulement
- Pretty printing `->` = $\rightarrow$, `forall` = $\forall$

`Type` VS `Prop`

termes		types		types (sortes)

```
          3          :          nat            : Type

       Nat.add       :   nat -> nat -> nat     : Type

       1 ?= 3        :         bool            : Type
```

```
     eq_refl 0       :         0 = 0           : Prop.

       (O_S 1)       :         0 <> 2          : Prop.

     Nat.le_0_1      :         0 <= 1          : Prop.

fun n:nat => eq_refl n:   forall (n:nat), n = n  : Prop.
```


- types contiennent des termes (types dépendants),
- typiquement les propriétés mais pas seulement
- Pretty printing `->` = $\rightarrow$, `forall` = $\forall$

`Prop`

L'usage principal d'un type `T:Prop` *est de savoir si on peut exhiber un terme de type* `T` *ou* `T → False` *(càd* `¬T`*).*

`Prop`

L'usage principal d'un type `T:Prop` est de savoir si on peut exhiber un terme de type `T` ou `T → False` (càd `¬T`).

preuve de `T` — réfutation de `T`

- seul l'un des deux est possible (cohérence de Coq (sans axiome))
- voire aucun (incomplétude des systèmes logiques)
- construire le terme
  - directement (`exact (fun x => ...).`)
  - ou à l'aide de tactiques `intro x. induction x. ...`
  - https://coq.inria.fr/files/coq-itp-2015/CoqSurvivalKit.pdf
  - https://coq.inria.fr/tutorial-nahas
  - https://coq.inria.fr/tutorial/2-induction

`Prop`

L'usage principal d'un type `T:Prop` *est de savoir si on peut exhiber un terme de type* `T` *ou* `T → False` *(càd* `¬T`*).*

preuve de `T`

réfutation de `T`

Ex :

- `le 3 4` (noté `3 <= 4`) a le type `Prop`.
  On peut exhiber `(le_S 3 3 (le_\n\ 3)) : 3 <= 4`.
- on ne peut pas exhiber de terme de type `le 4 3`.
- on peut exhiber `(Nat.nle_succ_diag_l 3) : le 4 3 → False` (ou `¬4 <= 3` ou `¬4 <= 3`).

Incomplétude et axiomes

- Ajouter une propriété indéterminée en axiome est cohérent.

```
Axiome TiersExclus: ∀A:Prop, A \/ ¬A.
```

```
Require Import Classical. Check classical.
```

- les axiomes précédemment indéterminés ne le sont peut-être plus maintenant !
- Ex (fameux mais obsolète car maintenant `Set` prédicatif) : Tiers exclus + axiome du choix
- Privilégier les librairies axiom-free (Laïcité = Liberté)
- https://github.com/coq/coq/wiki/The-Logic-of-Coq
- https://github.com/coq/coq/wiki/CoqAndAxioms

Prop $\neq$ bool

- Propriétés $\neq$ booléens
- `eq nat: nat → nat → Prop`
- `Nat.eqb: nat → nat → bool`
- `eq x 3` (aka `x = 3`) est une propriété : on peut essayer de la prouver
- `Nat.eqb x 3` (aka `x =? 3`) est une donnée booléenne qui se calcule

Prop $\neq$ bool

Remarques :

- dans `Prop` pas besoin d'être calculable ($\neq$ improuvable)
- si calculable alors peut être utile :

```
Lemma maPropriete_OK: ∀ a b c,
  maPropriete a b c = true ↔ Mapropriete a b c.
```

  ou carrément :

```
Definition MaPropriete a b c := maPropriete a b c = true.
```

Prop $\neq$ bool

Remarques :

- dans `Prop` pas besoin d'être calculable ($\neq$ improuvable)
- si calculable alors peut être utile :

```
Lemma maPropriete_OK: ∀ a b c,
 maPropriete a b c = true ↔ Mapropriete a b c.
```

ou carrément :

```
Definition MaPropriete a b c := maPropriete a b c = true.
```

Prop ≠ bool

Remarques :

- dans `Prop` pas besoin d'être calculable (≠ improuvable)
- si calculable alors peut être utile :

```
Lemma maPropriete_OK: ∀ a b c,
 maPropriete a b c = true ↔ Mapropriete a b c.
```

fonction de décision (bool)   propriété (Prop)

ou carrément :

```
Definition MaPropriete a b c := maPropriete a b c = true.
```

bool vs sumbool

Plus généralement si `P(x) \/ Q(x)` est décidable :

- Choix 1 :

```
Definition decidePQ x: bool := ... .
Lemma PorQdecP: ∀ x, decidePQ x = true → P x.
Lemma PorQdecQ: ∀ x, decidePQ x = false → Q x.
```

- Choix 2 : `sumbool` = `bool` embarquant la propriété décidée :

```
Lemma PorQdec: ∀ x, {P x} + {Q x}. (* ≡ sumbool (P x) (Q x) *)
```

bool vs sumbool

Plus généralement si `P(x) \/ Q(x)` est décidable :

- Choix 1 :

```
Definition decidePQ x: bool := ... .
Lemma PorQdecP: ∀ x, decidePQ x = true → P x.
Lemma PorQdecQ: ∀ x, decidePQ x = false → Q x.
```

- Choix 2 : `sumbool` = `bool` embarquant la propriété décidée :

```
Lemma PorQdec: ∀ x, {P x} + {Q x}. (* ≡ sumbool (P x) (Q x) *)
```

```
Inductive sumbool (A B : Prop) : Set :=
| left : A → {A} + {B}
| right : B → {A} + {B}
where "{ A } + { B }" := (sumbool A B) : type_scope.
```

(Voir types inductifs plus bas)

bool vs sumbool

```
Lemma PorQdec: ∀ x, {P x} + {Q x}.
```

bool vs sumbool

```
Lemma PorQdec: ∀ x, {P x} + {Q x}.
```

```
H: { maPropriete x } + { ¬ maPropriete x }
==================
...

destruct H.

2: subgoals
H: maPropriete x
==================
...

H: ¬ maPropriete n
==================
...
```

Définition de type

```
Definition T: T' := T''.

Inductive T: T' :=
| C1: T1 → T2 → T
| C2: ... → T.

Record T: T' := {
  fld1: T1;
  fld2: T2;
  ...
}.

Class T: T' :=  {
  fld1: T1;
  fld2: T2;
  ...
}.
```

Définition de valeur

```
Definition t: T := ... .

Definition c1: T := (C1 t1 t2).
Definition c2: T := (C2 ...)


Definition inst: T := {| fld1 := ...;
                         fld2 := ... |}.
 ... fld1 inst ...
 ... inst.(fld1) ...


Instance inst : T := {| fld1 := ...;
                        fld2 := ... |}.
 ... fld1 inst ...
 ... inst.(fld1) ...
```

Exemples

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

```
Definition Pair: Type := nat * nat.
```

```
Inductive Day: Type := Monday | Tuesday | Wednesday
    | Thursday | Friday | Saturday | Sunday.
```

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A
```

```
Definition oneone : Pair := (1,1).
Definition twotwo := (2,2). (* nat*nat ≡ Pair *)
Check (twotwo:Pair).
Definition d := Monday.
Definition workwk := cons Day Monday (cons Day Tuesday
 (cons Day Wednesday (cons Day Thursday (cons Day Friday (nil Day))))).
```

Exemples

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

```
Definition Pair: Type := nat * nat.
```

```
Inductive Day: Type := Monday | Tuesday | Wednesday
    | Thursday | Friday | Saturday | Sunday.
```

paramètre

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A
```

```
Definition oneone : Pair := (1,1).
Definition twotwo := (2,2). (* nat*nat ≡ Pair *)
Check (twotwo:Pair).
Definition d := Monday.
Definition workwk := cons Day Monday (cons Day Tuesday
 (cons Day Wednesday (cons Day Thursday (cons Day Friday (nil Day))))).
```

appliqué

Records dépendants = Modules

```
Record rcrd := {
  fld1 : Type;
  fld2 : nat;
}.





Definition x:= {| fld1:=bool; fld2:=3;|}




Check (fld1 x).
Check (x.(fld1)). (* alternate syntax *)
```

- = Inductif à 1 constructeur
- Accesseurs = fonctions de projection
- *Type d'un champ peut dépendre d'un champ précédent*

Records dépendants = Modules

```
Record rcrd := {
  fld1 : Type;
  fld2 : nat;
  fld3 : fld1;
  fld4 : nat → fld1;
  fld5 : ∀ x y:fdl1, x <> y → fld4 x <> fld4 y ;
}.

Definition x:= {| fld1:=bool; fld2:=3; fld3:=true;
                  fld4:= fun x:nat => Nat.even x;
                  fld5:= ... |}.

Check (fld1 x).
Check (x.(fld1)). (* alternate syntax *)
```

- = Inductif à 1 constructeur
- Accesseurs = fonctions de projection
- *Type d'un champ peut dépendre d'un champ précédent*

Types inductifs

Support de cours : Série Software Foundation (B. Pierce et al.)

- Livre : Logical Foundation
  - Chapitre ≪ Induction ≫
  - Chapitre ≪ Inductively defined propositions ≫
    - Section ≪ Inductively Defined Propositions ≫[1]
    - Section ≪ Using Evidence in Proofs ≫[1]

1. Attention : url fragiles

Types inductifs

Support de cours : Série Software Foundation (B. Pierce et al.)

- Livre : Logical Foundation
  - Chapitre « Induction »
  - Chapitre « Inductively defined propositions »
    - Section « Inductively Defined Propositions »[1] lire
    - Section « Using Evidence in Proofs »[1] lire

1. Attention : url fragiles

Définition inductive

Trois façons de définir un ensemble :

- par extension : $\{1, 2, 3\}$, $\{0, S(0), S(S(0))\}$
- par intention : $\{n \in \mathbb{N} | n \leq 3\}$
  (remarque : à partir d'un ensemble déjà défini (ici $\mathbb{N}$))
- par induction

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \rightarrow U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

Tous les arguments ne sont pas récursifs
Certains ne sont même pas dans U

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E$ = plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

Arguments récursifs — Arguments non récursifs

$U = \mathbb{N}$
$B = \{0\} \subset \mathbb{N}$

$$\Omega = \left\{ \begin{array}{lccc} c_1 : & \mathbb{N} & \to & \mathbb{N} \\ & n & \mapsto & n+2 \end{array} \right\}$$

$\{0, 2, 4, 6 \dots\}$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

? $\{1, 3, 5, 7 \dots\}$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E$ = plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

$$U = \mathbb{N}$$
$$B = \{1\} \subset \mathbb{N}$$
$$\Omega = \left\{ \begin{array}{cccc} c_1 : & \mathbb{N} & \to & \mathbb{N} \\ & n & \mapsto & n+2 \end{array} \right\}$$

$$\{1, 3, 5, 7 \dots\}$$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \ldots c_n\}$ <u>opérateurs</u>, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E <u>clos</u> par Ω

$U = \mathbb{N} \times \mathbb{N}$

$B = \{(n, n) \in \mathbb{N} \times \mathbb{N}\}$ (intention)

$$\Omega = \left\{ \begin{array}{llll} c_1 : & \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \times \mathbb{N} \\ & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

?

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

$U = \mathbb{N} \times \mathbb{N}$

$B = \{(n, n) \in \mathbb{N} \times \mathbb{N}\}$ (intention)

$$\Omega = \left\{ \begin{array}{llll} c_1 : & \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \times \mathbb{N} \\ & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{lll} (0,0), & (0,1), & (0,2), \dots \\ & (1,1) & (1,2), \dots \\ & & (2,2), \dots \end{array} \right\}$$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

$$U = \Sigma^* \text{ with } \Sigma = \{a, b, c\}$$
$$B = \{\epsilon\}$$
$$\Omega = \left\{ \begin{array}{llcl} c_1 : & \Sigma^* \times \{a, b\} & \to & \Sigma^* \\ & (w, x) & \mapsto & xwx \\ c_2 : & \Sigma^* \times \Sigma^* & \to & \Sigma^* \\ & (w, w') & \mapsto & wcw' \end{array} \right\}$$

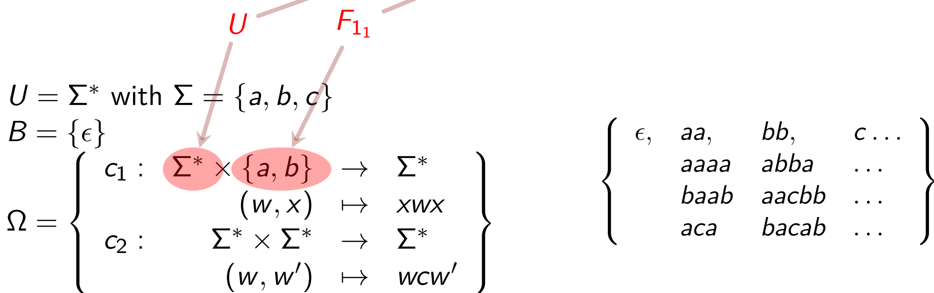
$$\left\{ \begin{array}{llll} \epsilon, & aa, & bb, & c \ldots \\ & aaaa & abba & \ldots \\ & baab & aacbb & \ldots \\ & aca & bacab & \ldots \end{array} \right\}$$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω



Induction (maths) : définir $E \subset U$ (U déjà défini)

Pour définir E il faut

1. $B \subset U$ éléments de base (B défini par intention, extension ou induction)
2. $\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E$ = plus petit ensemble t.q. : (1) $B \subset E$ et (2) E clos par Ω

$p_i = 0$

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E$ = plus petit ensemble clos par Ω.

Induction (maths) : définir $E \subset U$ (U déjà défini)

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$
$\Rightarrow E =$ plus petit ensemble clos par Ω.

fonction à 0 arguments

$$\Omega = \left\{ \begin{array}{ll} c_1 : & 0 \\ c_2 : & n \mapsto n+2 \end{array} \right\}$$

$\{0, 2, 4, 6 \ldots\}$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \rightarrow U$
$\Rightarrow E =$ plus petit ensemble clos par Ω.

$U = \mathbb{N} \times \mathbb{N}$

$$\Omega = \left\{ \begin{array}{llll} c_1 : & & ? & \\ & & & \\ c_2 : & \mathbb{N} \times \mathbb{N} & \rightarrow & \mathbb{N} \times \mathbb{N} \\ & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{lll} (0,0), & (0,1), & (0,2), \dots \\ & (1,1) & (1,2), \dots \\ & & (2,2), \dots \end{array} \right\}$$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$
$\Rightarrow E =$ plus petit ensemble clos par Ω.

$$U = \mathbb{N} \times \mathbb{N}$$

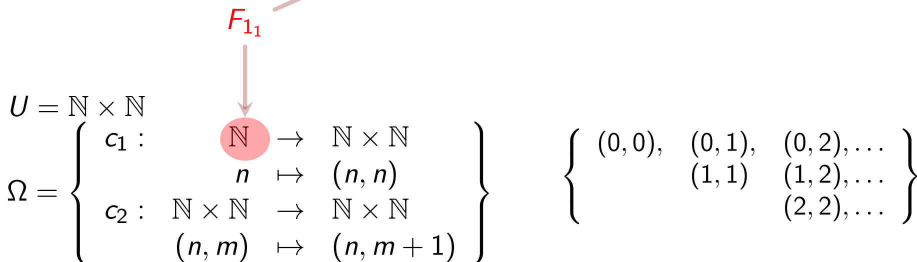
$$\Omega = \left\{ \begin{array}{llll} c_1 : & & ? & \\ & & & \\ c_2 : & \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \times \mathbb{N} \\ & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{lll} (0,0), & (0,1), & (0,2), \dots \\ & (1,1) & (1,2), \dots \\ & & (2,2), \dots \end{array} \right\}$$

Induction (maths) : définir $E \subset U$ (U déjà défini)

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \ldots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$
$\Rightarrow E$ = plus petit ensemble clos par Ω.



Induction (maths) : définir $E \subset U$ (U déjà défini)

Autre formulation : uniquement Ω

$\Omega = \{c_1, c_2 \dots c_n\}$ opérateurs, où $c_i : U^{p_i} \times F_{i_1} \cdots \times F_{i_{q_i}} \to U$

$\Rightarrow E =$ plus petit ensemble clos par Ω.

F_{1_1}

$U^0 \times \mathbb{N}$

$$U = \mathbb{N} \times \mathbb{N}$$

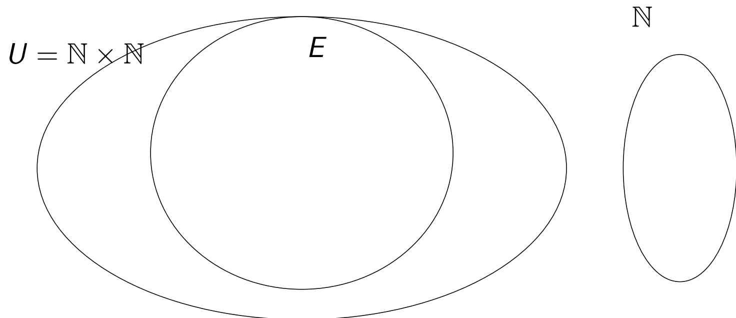
$$\Omega = \left\{ \begin{array}{lrcl} c_1 : & \mathbb{N} & \to & \mathbb{N} \times \mathbb{N} \\ & n & \mapsto & (n, n) \\ c_2 : & \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \times \mathbb{N} \\ & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{lll} (0,0), & (0,1), & (0,2), \dots \\ & (1,1) & (1,2), \dots \\ & & (2,2), \dots \end{array} \right\}$$

Induction

$$\Omega = \left\{ \begin{array}{llll} c_1: & n & \mapsto & (n, n) \\ c_2: & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

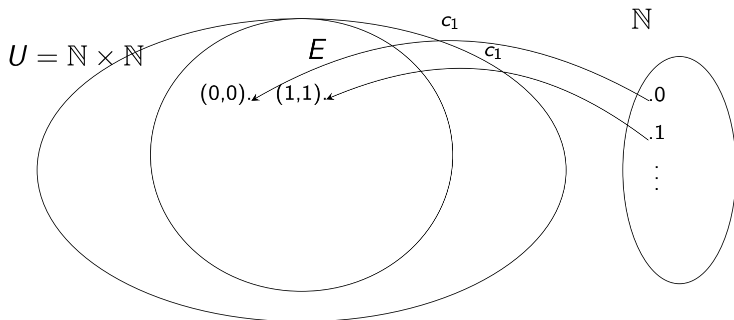
$$\left\{ \begin{array}{r} \cdots \\ \cdots \\ \cdots \end{array} \right\}$$



Induction

$$\Omega = \left\{ \begin{array}{llll} c_1 : & n & \mapsto & (n, n) \\ c_2 : & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{llll} (0,0), & & & \dots \\ & (1,1), & & \dots \\ & & (2,2), & \dots \end{array} \right\}$$

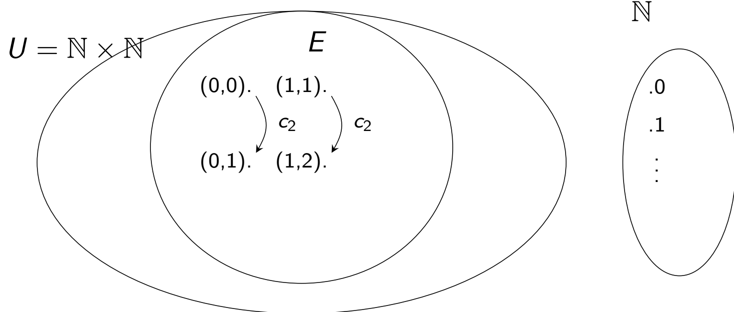


Induction

$$\Omega = \left\{ \begin{array}{lrcl} c_1 : & n & \mapsto & (n,n) \\ c_2 : & (n,m) & \mapsto & (n,m+1) \end{array} \right\}$$

$c2(c_1(0))$ $c_2(c_1(1))$

$$\left\{ \begin{array}{llll} (0,0), & (0,1), & & \dots \\ & (1,1), & (1,2), & \dots \\ & & (2,2), & \dots \end{array} \right\}$$

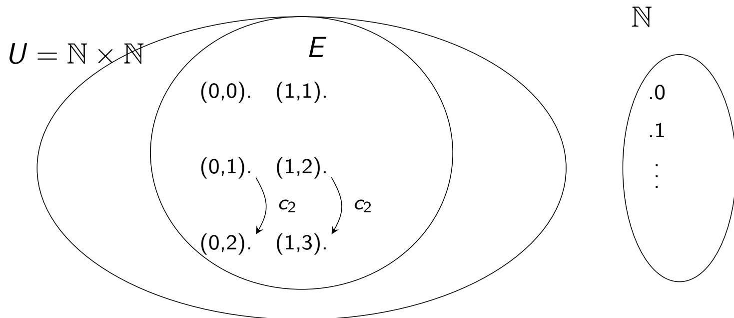


Induction

$$\Omega = \left\{ \begin{array}{llll} c_1 : & n & \mapsto & (n, n) \\ c_2 : & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$c_2(c_2(c_1(0)))$

$$\left\{ \begin{array}{llll} (0,0), & (0,1), & (0,2), & \dots \\ & (1,1), & (1,2), & \dots \\ & & (2,2), & \dots \end{array} \right\}$$



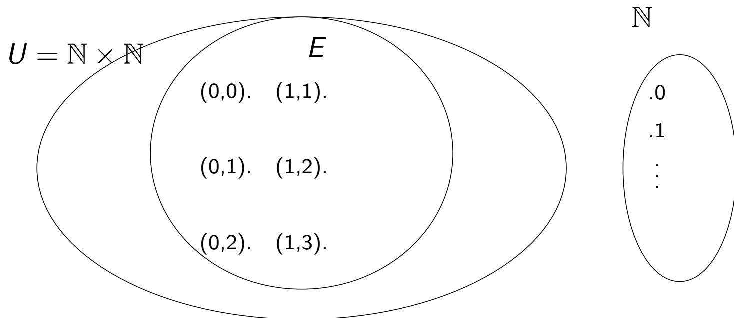
$\mathbb{N}$

.0
.1
⋮

Induction

$$\Omega = \left\{ \begin{array}{llll} c_1 : & n & \mapsto & (n, n) \\ c_2 : & (n, m) & \mapsto & (n, m+1) \end{array} \right\}$$

$$\left\{ \begin{array}{llll} (0,0), & (0,1), & (0,2), & \ldots \\ & (1,1), & (1,2), & \ldots \\ & & (2,2), & \ldots \end{array} \right\}$$



Induction opérateurs vs induction constructeurs

- il faut définir un U d'abord
- égalité possible entre 2 application différentes des opérateurs :

$$c_i(...) = c_j(...)$$

- Cas particulier :
  - U l'ensemble de tous les termes clos sur n'importe quelle algèbre.

$$U = \{t | \exists S, t \in \mathcal{T}(S, \emptyset)\}$$

  - $c_i(...)$ opérateur retournant le terme $c_i(...)$ lui-même

Induction (Constructeurs)

Définir un ensemble E de termes « frais »

- $\Omega = \{c_1, c_2 \ldots c_n\}$ constructeur, $c_i : E \times ... \times E \times A_1 \cdots \times A_m \to E$

E est le plus petit ensemble clos par Ω.

Propriétés

- chaque $c_i(b_1 ... b_n, a_1 ... a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là.

Induction (Constructeurs)

Définir un ensemble E de termes « frais »

- $\Omega = \{c_1, c_2 \ldots c_n\}$ constructeur, $c_i : E \times \ldots \times E \times A_1 \cdots \times A_m \to E$

E est le plus petit ensemble clos par Ω.

Propriétés

- chaque $c_i(b_1 \ldots b_n, a_1 \ldots a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là.

Bref : algèbre des termes clos $\mathcal{T}(\{c_1 \ldots c_n\}, \emptyset)$

Induction (constructeur)

$$\Omega = \{\texttt{O} : E,\ \texttt{S} : E \rightarrow E\}$$

```
Inductive nat : Set :=
  O : nat
| S : nat → nat.
```

$$\Omega = \{\texttt{nil} : E,\ \texttt{cons} : nat \times E \rightarrow E\}$$

```
Inductive listnat : Type :=
  nil : listnat
| cons : nat → listnat → listnat.
```

$$\Omega = \{\texttt{xH} : E,\ \texttt{xI} : E \rightarrow E,\ \texttt{xO} : E \rightarrow E\}$$

```
Inductive positive : Set :=
  xI : positive → positive  (*1w*)
| xO : positive → positive  (*0w*)
| xH : positive.            (* 1 *)
```

$$\Omega = \{\texttt{Z0} : E,\ \texttt{Zpos} : positive \rightarrow E,\ \texttt{Zneg} : positive \rightarrow E\}$$

```
Inductive Z : Set :=
  Z0 : Z
| Zpos : positive → Z
| Zneg : positive → Z.
```

Induction(constructeurs)

$$\Omega = \{\texttt{Leaf} : nat \to E,\ \texttt{Node} : nat \times E \times E \to E\}$$

```
Inductive Tree: Type :=
|Leaf: nat → Tree
|Node: nat → Tree → Tree → Tree.
```

$$\{1,\ \begin{array}{c}2\\ 3\quad 4\end{array},\ \begin{array}{c}2\\ 3\quad 4\\ \quad\ \ 7\ 5\end{array},\ \ldots\}$$

Induction(constructeur)

AST d'un langage de programmation (imp) ?

Induction(constructeur)

AST d'un langage de programmation (imp) ?

$$\Omega = \{\texttt{Cskip} : E,\ \texttt{CAss} : \mathit{string} \times \mathit{aexp} \to E,\ \texttt{CSeq} : E \times E \to E,\ \texttt{CIf} : \mathit{bexp} \times E \times E \to E,\ \texttt{CWhile} : \mathit{bexp} \times E \to E\ \}$$

```
Inductive com : Type :=
| CSkip : com
| CAss : string → aexp → com
| CSeq : com → com → com
| CIf : bexp → com → com → com
| CWhile : bexp → com → com.
```

Induction(constructeur)

AST d'un langage de programmation (imp) ?

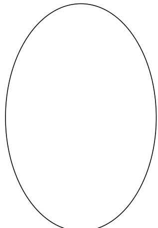
$$\Omega = \{\texttt{Cskip} : E,\ \texttt{CAss} : \mathit{string} \times \mathit{aexp} \to E,\ \texttt{CSeq} : E \times E \to E,\ \texttt{CIf} : \mathit{bexp} \times E \times E \to E,\ \texttt{CWhile} : \mathit{bexp} \times E \to E\ \}$$

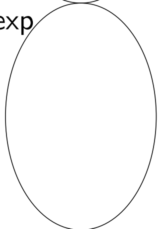
```
Inductive com : Type :=
| CSkip : com
| CAss : string → aexp → com
| CSeq : com → com → com
| CIf : bexp → com → com → com
| CWhile : bexp → com → com.
```

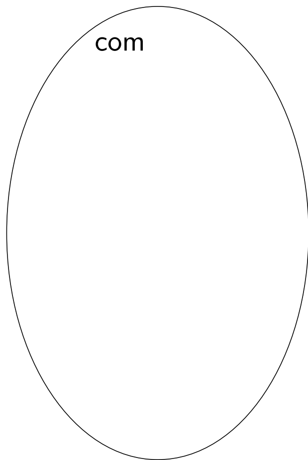
```
if (...)
  { x = ...; }
else
  { y = ...; }
z  = ...;
```

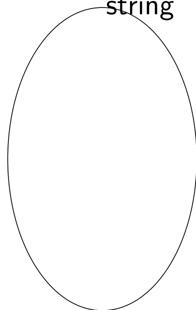
```
CSeq(
  CIf(...,
    CAss("x",...),
    CAss("y",...)),
  CAss("z",...))
```

imp

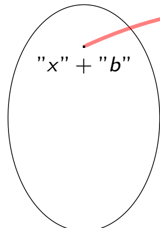


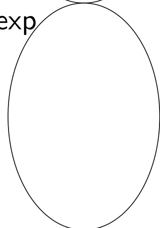


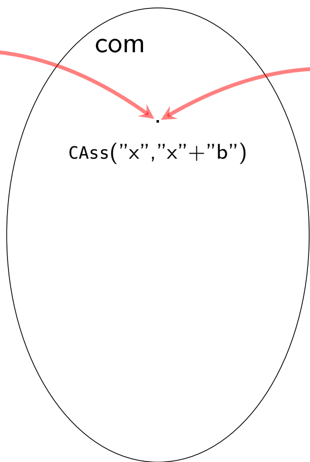


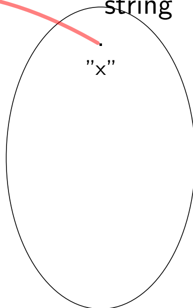


imp



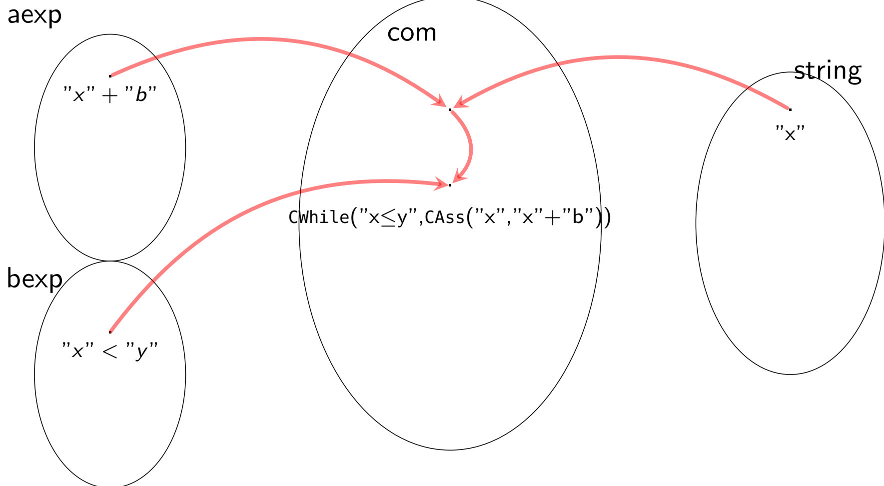






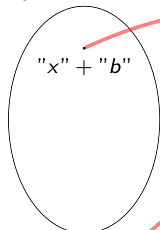

```
CAss : string → aexp → com
```

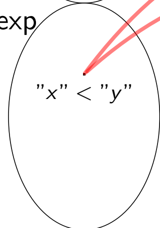
imp

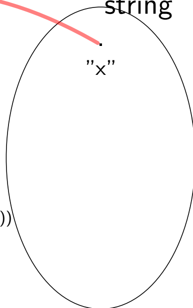



```
CWhile : bexp → com → com
```

imp



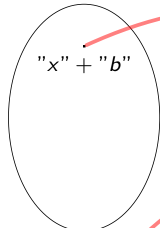




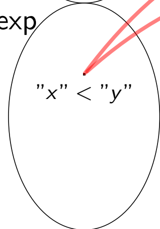

```
CIf : bexp → com → com → com
```

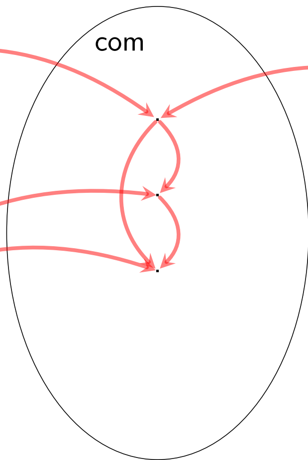
imp

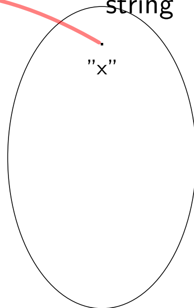
aexp



bexp







bexp ? aexp ?

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1 ... b_n, a_1 ... a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là $(+B)$.

$\Rightarrow$ pour définir $f : E \to F$, il suffit de décrire le résultat $f(x)$ pour :

- $x = c_i(y_1 ... y_n, a_1 ... a_m)$ pour tous les c_i, pour tout $\vec{y_i}$ et $\vec{a_j}$.
- en utilisant $f(y_1) ... f(y_n)$ (résultat d'appels récursifs)
- $+$ terminaison pour tout $c_i(y_1 ... y_n, a_1 ... a_m)$

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1...b_n, a_1...a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là $(+B)$.

Exhaustivité

$\Rightarrow$ pour définir $f : E \rightarrow F$, il suffit de décrire le résultat $f(x)$ pour :

- $x = c_i(y_1...y_n, a_1...a_m)$ pour tous les c_i, pour tout $\vec{y_i}$ et $\vec{a_j}$.
- en utilisant $f(y_1)...f(y_n)$ (résultat d'appels récursifs)
- $+$ terminaison pour tout $c_i(y_1...y_n, a_1...a_m)$

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1...b_n, a_1...a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là ($+B$).

Exhaustivité

$\Rightarrow$ pour définir $f : E \rightarrow F$, il suffit de décrire le résultat $f(x)$ pour :

- $x = c_i(y_1...y_n, a_1...a_m)$ pour tous les c_i, pour tout $\vec{y_i}$ et $\vec{a_j}$.
- en utilisant $f(y_1)...f(y_n)$ (résultat d'appels récursifs)
- $+$ terminaison pour tout $c_i(y_1...y_n, a_1...a_m)$

Récursion

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1 ... b_n, a_1 ... a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là ($+B$).

$\Rightarrow$ pour prouver $\forall x \in E, P(x)$,

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1...b_n, a_1...a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là ($+B$).

$\Rightarrow$ pour prouver $\forall x \in E, P(x)$, il suffit de prouver $P(x)$ pour :

- $x = c_i(y_1...y_n, a_1...a_m)$ pour tous les c_i, pour tous les $\vec{y_i}$ et $\vec{a_j}$.
- en utilisant $P(y_1)...P(y_n)$ (hypothèses d'induction)
- $+$ terminaison pour tout $c_i(y_1...y_n, a_1...a_m)$

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

Propriétés

- chaque $c_i(b_1 ... b_n, a_1 ... a_m)$ = nouvel élément de E, $\neq$ de tous les autres
- chacun sa propre valeur (ne représente pas un calcul $\neq$ opérateur)
- pas d'autres éléments que ceux-là ($+B$).

Exhaustivité

$\Rightarrow$ pour prouver $\forall x \in E, P(x)$, il suffit de prouver $P(x)$ pour :

- $x = c_i(y_1 ... y_n, a_1 ... a_m)$ pour tous les c_i, pour tous les $\vec{y_i}$ et $\vec{a_j}$.
- en utilisant $P(y_1) ... P(y_n)$ (hypothèses d'induction)
- + terminaison pour tout $c_i(y_1 ... y_n, a_1 ... a_m)$

Récursion

fonction récursive $\equiv$ démonstration par récurrence

Schéma d'induction (constructeurs)

- exhaustivité = pas d'élément en dehors des constructeurs
- récursion = Chaque $c_i(\vec{e_i}, \vec{a_i}) \in E$ = arbre fini, donc construit à partir :
  - de e_i « plus petit »
  - de a_i « en dehors » de E

```
Inductive nat : Set :=
 | O : nat                                        (* B = {O} *)
 | S : nat → nat                                  (* Ω = {c₁ = S} *)

Definition fact :=
  fix fact (n : nat) : nat :=
    match n with
    | 0 => 1
    | S n0 => S n0 * fact n0
    end.
```

Schéma d'induction (constructeurs)

- exhaustivité = pas d'élément en dehors des constructeurs
- récursion = Chaque $c_i(\vec{e_i}, \vec{a_i}) \in E$ = arbre fini, donc construit à partir :
  - de e_i « plus petit »
  - de a_i « en dehors » de E

```
Inductive nat : Set :=
 | O : nat                                    (* B = {0} *)
 | S : nat → nat                        (* Ω = {c1 = S} *)

Definition fact :=
 fix fact (n : nat) : nat :=
   match n with
    | 0 => 1
    | S n0 => S n0 * fact n0
   end.
```

Récursion

Exhaustivité

Schéma d'induction (constructeurs)

Principe d'induction prouvable en Coq (certains automatiquement)

```
Inductive nat : Set :=
 | O : nat                                        (*  B = {O}           *)
 | S : nat → nat                                  (*  Ω = {c1 = S}      *)

Check nat_ind.
```

```
 ∀ P : nat → Prop,
  P 0                                             (*  x ∈ B                 *)
  → (∀ n : nat, P n → P (S n))                    (*  P(y1) → P(c1(y1))     *)
  → ∀ n : nat, P n
```

```
Print nat_ind.
```

```
  fun (P : nat → Prop) (f : P 0) (f0 : _) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                         | 0 => f
                         | S n0 => f0 n0 (F n0)
                         end
```

Schéma d'induction (constructeurs)

Principe d'induction prouvable en Coq (certains automatiquement)

```
Inductive nat : Set :=
 | O : nat                       (* B = {O}        *)
 | S : nat → nat                 (* Ω = {c1 = S}   *)

Check nat_ind.
```

```
∀ P : nat → Prop,
  P 0                                    (* x ∈ B                 *)
  → (∀ n : nat, P n → P (S n))           (* P(y1) → P(c1(y1))     *)
  → ∀ n : nat, P n
```

```
Print nat_ind.
```

Exhaustivité

```
  fun (P : nat → Prop) (f : P 0) (f0 : _) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                          | 0 => f
                          | S n0 => f0 n0 (F n0)
                         end
```

Récursion

Schéma d'induction (constructeurs)

Principe d'induction prouvable en Coq (certains automatiquement)

```
Inductive nat : Set :=
 | O : nat                                    (* B = {O}           *)
 | S : nat → nat                              (* Ω = {c1 = S}      *)

Check nat_ind.
```

```
 ∀ P : nat → Prop,
  P 0                                         (* x ∈ B                   *)
  → (∀ n : nat, P n → P (S n))                (* P(y1) → P(c1(y1))       *)
  → ∀ n : nat, P n
```

Commentaires mathématiques : $B = \{O\}$; $\Omega = \{c_1 = S\}$; $x \in B$; $P(y_1) \to P(c_1(y_1))$

```
Print nat_ind.
```

```
  fun (P : nat → Prop) (f : P 0) (f0 : _) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                         | 0 => f
                         | S n0 => f0 n0 (F n0)
                         end
```

Schéma d'induction (constructeurs)

```
com_ind
     : ∀ P : com → Prop,
        P CSkip →
        (∀ (s : string) (a : aexp), P (CAss s a)) →
        (∀ c : com, P c → ∀ c0 : com, P c0 → P (CSeq c c0)) →
        (∀ (b : bexp) (c : com),
           P c → ∀ c0 : com, P c0 → P (CIf b c c0)) →
        (∀ (b : bexp) (c : com), P c → P (CWhile b c)) →
       ∀ c : com, P c
```

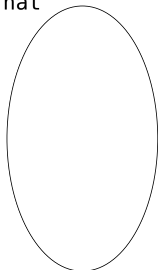
Schéma d'induction (constructeurs)

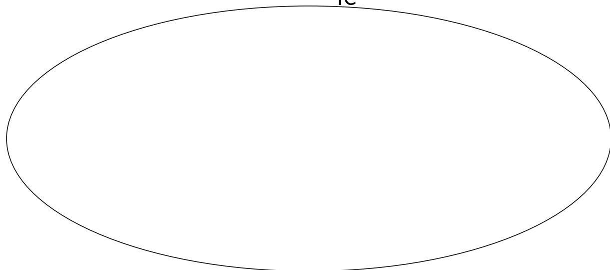
```
fun (P : com → Prop) (f : P CSkip)
  (f0 : ∀ (s : string) (a : aexp), P (CAss s a))
  (f1 : ∀ c : com, P c → ∀ c0 : com, P c0 → P (CSeq c c0))
  (f2 : ∀ (b : bexp) (c : com),
        P c → ∀ c0 : com, P c0 → P (CIf b c c0))
  (f3 : ∀ (b : bexp) (c : com), P c → P (CWhile b c)) =>
fix F (c : com) : P c :=
  match c as c0 return (P c0) with
  | CSkip => f
  | CAss s a => f0 s a
  | CSeq c0 c1 => f1 c0 (F c0) c1 (F c1)
  | CIf b c0 c1 => f2 b c0 (F c0) c1 (F c1)
  | CWhile b c0 => f3 b c0 (F c0)
  end
```

Constructeurs + types dépendant = opérateurs

- Si E : *Prop* défini par induction : ensemble de ~~valeurs~~ preuves
- Exemple [2] :

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n
 | le_S : ∀ n m : nat, le n m → le n (S m).
```



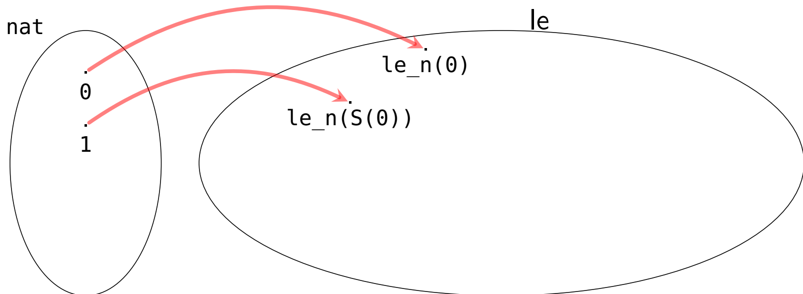


2. Dans la bibliothèque standard n est en paramètre

Constructeurs + types dépendant = opérateurs

- Si E : *Prop* défini par induction : ensemble de ~~valeurs~~ preuves
- Exemple[2] :

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n
 | le_S : ∀ n m : nat, le n m → le n (S m).
```

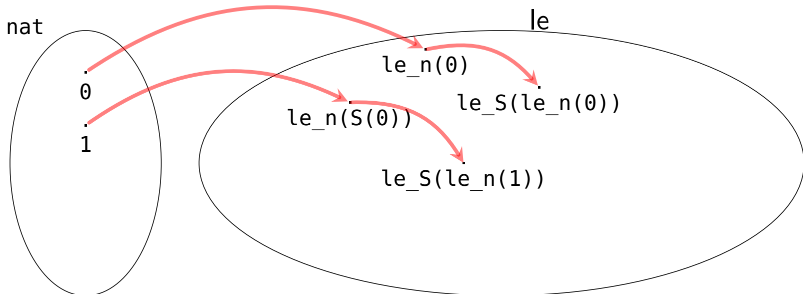


2. Dans la bibliothèque standard n est en paramètre

Constructeurs + types dépendant = opérateurs

- Si E : *Prop* défini par induction : ensemble de ~~valeurs~~ preuves
- Exemple[2] :

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n
 | le_S : ∀ n m : nat, le n m → le n (S m).
```

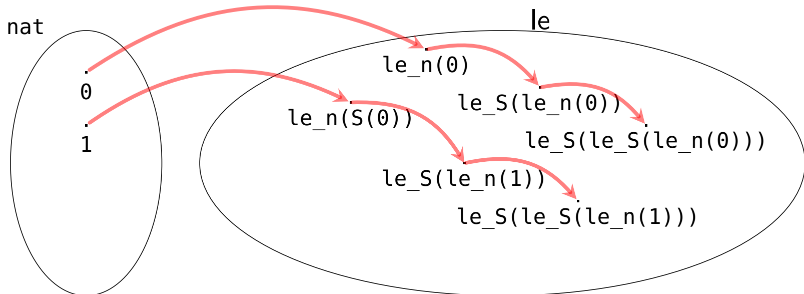


2. Dans la bibliothèque standard n est en paramètre

Constructeurs + types dépendant = opérateurs

- Si *E* : *Prop* défini par induction : ensemble de ~~valeurs~~ preuves
- Exemple[2] :

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n
 | le_S : ∀ n m : nat, le n m → le n (S m).
```

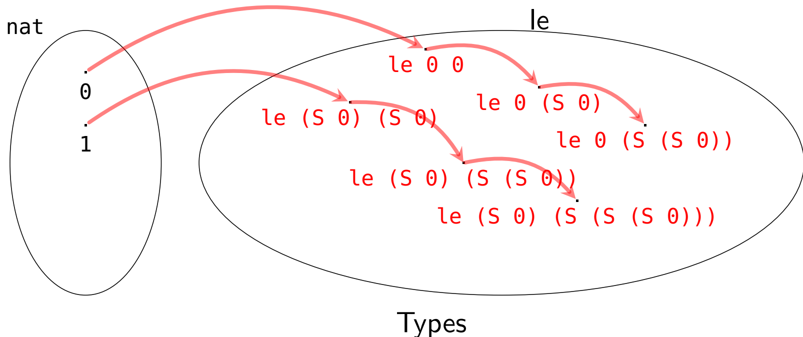


2. Dans la bibliothèque standard *n* est en paramètre

Constructeurs + types dépendant = opérateurs

- Si E : *Prop* défini par induction : ensemble de ~~valeurs~~ preuves
- Exemple[2] :

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n
 | le_S : ∀ n m : nat, le n m → le n (S m).
```



2. Dans la bibliothèque standard n est en paramètre

Constructeurs + types dépendant = règles d'inférences

```
Inductive le : nat → nat → Prop :=
 | le_n : ∀ n : nat, le n n                        (*   ------   *)
                                                   (*   n ≤ n    *)
 | le_S : ∀ n m : nat, le n m → le n (S m).        (*   n ≤ m    *)
                                                   (* --------   *)
                                                   (* n ≤ S(m)   *)
```

Sémantique de Imp

```
Inductive ceval : com → state → state → Prop :=
| E_Skip : ∀ st,
   st =[ SKIP ]=> st                          (*  ─────────────────  *)
                                              (*  SKIP, σ ⇝ σ        *)
| E_Ass  : ∀ st a1 n x,
   aeval st a1 = n →                          (*  σ, a1 →a n          *)
   st =[ x ::= a1 ]=> (x !→ n ; st)           (*  ─────────────────── *)
                                              (*  x:=a1, σ ⇝ σ[x ← n] *)
| E_Seq : ∀ c1 c2 st st' st'',
   st  =[ c1 ]=> st'  →                       (*  c1, σ ⇝ σ'          *)
   st' =[ c2 ]=> st'' →                       (*  c2, σ' ⇝ σ''        *)
   st  =[ c1 ;; c2 ]=> st''                   (*  ─────────────────── *)
                                              (*  c1;c2, σ ⇝ σ''      *)
| E_IfTrue : ∀ st st' b c1 c2,
   beval st b = true →                        (*  σ, b →b true        *)
   st =[ c1 ]=> st' →                         (*  c1, σ ⇝ σ'          *)
   st =[TEST b THEN c1 ELSE c2 FI]=> st'      (*  ──────────────────────────────── *)
                                              (*  if b then c1 else c2, σ ⇝ σ'     *)
...
  where "st =[ c ]=> st'" := (ceval c st st').
```

Sémantique de Imp

```
| E_IfFalse : ∀ st st' b c1 c2,
  beval st b = false →                    (*        σ, b →_b false        *)
  st =[ c2 ]=> st' →                      (*         c₂, σ ⇝ σ'           *)
  st =[TEST b THEN c1 ELSE c2 FI]=> st'   (*------------------------------*)
                                          (*  if b then c₁ else c₂, σ ⇝ σ' *)
| E_WhileFalse : ∀ b st c,
  beval st b = false →                    (*        σ, b →_b false        *)
  st =[ WHILE b DO c END ]=> st           (*   -----------------------    *)
                                          (*    while b do c, σ ⇝ σ       *)
| E_WhileTrue : ∀ st st' st'' b c,
  beval st b = true →                     (*        σ, b →_b true         *)
  st  =[ c ]=> st' →                      (*          c, σ ⇝ σ'           *)
  st' =[ WHILE b DO c END ]=> st'' →      (*    while b do c, σ' ⇝ σ''    *)
  st  =[ WHILE b DO c END ]=> st''        (*   ------------------------   *)
                                          (*    while b do c, σ ⇝ σ''     *)

  where "st =[ c ]=> st'" := (ceval c st st').
```

Constructeurs + types dépendants

```
...
| E_Ass  : ∀ σ a1 n x,
    aeval σ a1 = n  →  (x ::= a1) / σ \\ σ & { x !→ n }
| E_Seq : ∀ c1 c2 σ σ' σ'',
    c1 / σ  \\ σ'  →  c2 / σ' \\ σ''  →  (c1 ;; c2) / σ \\ σ''
| E_IfFalse : ∀ σ σ' b c1 c2,
    beval σ b = false  →  c2 / σ \\ σ' →  (IFB b THEN c1 ELSE c2 FI) / σ \\ σ'
```

```
E_Seq (X ::= 2) (IFB X <= 1 THEN Y::=3 ELSE Z::=4) { !→ 0} {X !→ 2} {X !→ 2; Z !→ 4}
 (E_Ass { !→ 0} 2 2 X (@eq_refl nat 2))
 (E_IfFalse {X !→ 2} {X !→ 2; Z !→ 4} (X <= 1) (Y::=3) (Z::=4) (@eq_refl bool false)
   (E_Ass {X !→ 2} 4 4 Z (@eq_refl nat 4)))  :   ?
```

$$\dfrac{\text{E_Ass}\ \dfrac{2 \rightarrow_a 2}{x{:=}2,\ \emptyset \leadsto \{("X",2)\}} \qquad \dfrac{\texttt{X <= 1} \rightarrow_b \texttt{false} \qquad \dfrac{4 \rightarrow_a 4}{\texttt{Z := 4},\ \{("X",2)\} \leadsto ?}\ \text{E_Ass}}{\texttt{if X <= 1 then...else Z := 4},\ \{("X",2)\} \leadsto ?}\ \text{E_IfFalse}}{?}\ \text{E_Seq}$$

Constructeurs + types dépendants

```
...
| E_Ass  : ∀ σ a1 n x,
   aeval σ a1 = n  →  (x ::= a1) / σ \\ σ & { x !→ n }
| E_Seq : ∀ c1 c2 σ σ' σ'',
   c1 / σ  \\ σ'  →  c2 / σ' \\ σ''  →  (c1 ;; c2) / σ \\ σ''
| E_IfFalse : ∀ σ σ' b c1 c2,
   beval σ b = false  →  c2 / σ \\ σ' →  (IFB b THEN c1 ELSE c2 FI) / σ \\ σ'
```

```
E_Seq (X ::= 2) (IFB X <= 1 THEN Y::=3 ELSE Z::=4) { !→ 0} {X !→ 2} {X !→ 2; Z !→ 4}
 (E_Ass { !→ 0} 2 2 X (@eq_refl nat 2))
 (E_IfFalse {X !→ 2} {X !→ 2; Z !→ 4} (X <= 1) (Y::=3) (Z::=4) (@eq_refl bool false)
   (E_Ass {X !→ 2} 4 4 Z (@eq_refl nat 4)))  :   ?
```

$$\dfrac{\dfrac{2 \rightarrow_a 2}{x{:=}2,\ \emptyset\ \rightsquigarrow\ \{(\text{"}X\text{"},2)\}}\ \text{E_Ass} \qquad \dfrac{\texttt{X <= 1} \rightarrow_b \texttt{false} \qquad \dfrac{4 \rightarrow_a 4}{\texttt{Z := 4},\ \{(\text{"}X\text{"},2)\}\ \rightsquigarrow ?}\ \text{E_Ass}}{\texttt{if X <= 1 then...else Z := 4},\ \{(\text{"}X\text{"},2)\}\ \rightsquigarrow ?}\ \text{E_IfFalse}}{?}\ \text{E_Seq}$$

Constructeurs + types dépendants

```
...
| E_Ass  : ∀ σ a1 n x,
    aeval σ a1 = n  →  (x ::= a1) / σ \\ σ & { x !→ n }
| E_Seq : ∀ c1 c2 σ σ' σ'',
    c1 / σ  \\ σ'  →  c2 / σ' \\ σ''  →  (c1 ;; c2) / σ \\ σ''
| E_IfFalse : ∀ σ σ' b c1 c2,
    beval σ b = false  →  c2 / σ \\ σ' →  (IFB b THEN c1 ELSE c2 FI) / σ \\ σ'
```

```
E_Seq (X ::= 2) (IFB X <= 1 THEN Y::=3 ELSE Z::=4) { !→ 0} {X !→ 2} {X !→ 2; Z !→ 4}
 (E_Ass { !→ 0} 2 2 X (@eq_refl nat 2))
 (E_IfFalse {X !→ 2} {X !→ 2; Z !→ 4} (X <= 1) (Y::=3) (Z::=4) (@eq_refl bool false)
   (E_Ass {X !→ 2} 4 4 Z (@eq_refl nat 4)))  :   ?
```

$$\dfrac{\text{E_Ass}\ \dfrac{2 \to_a 2}{x{:=}2,\ \emptyset \leadsto \{(\text{"}X\text{"}, 2)\}} \qquad \dfrac{\texttt{X <= 1} \to_b \texttt{false} \qquad \dfrac{4 \to_a 4}{\texttt{Z} := 4,\ \{(\text{"}X\text{"}, 2)\} \leadsto ?}\ \text{E_Ass}}{\texttt{if X <= 1 then...else Z} := 4,\ \{(\text{"}X\text{"}, 2)\} \leadsto ?}\ \text{E_IfFalse}}{?}\ \text{E_Seq}$$

Constructeurs + types dépendants

```
...
| E_Ass  : ∀ σ a1 n x,
    aeval σ a1 = n  →  (x ::= a1) / σ \\ σ & { x !→ n }
| E_Seq : ∀ c1 c2 σ σ' σ'',
    c1 / σ  \\ σ'  →  c2 / σ' \\ σ''  →  (c1 ;; c2) / σ \\ σ''
| E_IfFalse : ∀ σ σ' b c1 c2,
    beval σ b = false  →  c2 / σ \\ σ' →  (IFB b THEN c1 ELSE c2 FI) / σ \\ σ'
```

```
E_Seq (X ::= 2) (IFB X <= 1 THEN Y::=3 ELSE Z::=4) { !→ 0} {X !→ 2} {X !→ 2; Z !→ 4}
 (E_Ass { !→ 0} 2 2 X (@eq_refl nat 2))
 (E_IfFalse {X !→ 2} {X !→ 2; Z !→ 4} (X <= 1) (Y::=3) (Z::=4) (@eq_refl bool false)
   (E_Ass {X !→ 2} 4 4 Z (@eq_refl nat 4)))  :   ?
```

```
     : (X ::= 2;; IFB X <= 1 THEN Y ::= 3 ELSE Z ::= 4 FI) /
       { !→ 0} \\ {X !→ 2; Z !→ 4}
```

Constructeurs + types dépendants

- constructeurs = règles d'inférence
- constructeurs appliqués = règles d'inférence instanciées
- → types = propriété prouvée par les règles d'inférence instanciées
- Raisonnements par inductions/inversion sur les règles

```
 H : (X ::= 2;; IFB X <= 1 THEN Y ::= 3 ELSE Z ::= 4 FI) / { !→ 0} \\ {X !→ 2; Z !→ 4}
....
===================
....

inversion H.     (*   Énumérer les formes possibles à profondeur 1 de la preuve de H   *)
induction H.     (*            Raisonnement par induction sur la preuve de H            *)
```

Induction

Pour une induction réussie :

- Que des variables. Au besoin remplacer `P (C x)` par `∀y, y = C x → P y`
- Sur une propriété suffisamment générale pour être « inductive »
- Plutôt sur la propriété inductive que x doit respecter que sur x lui-même.

```
x:nat
H: even x
....
===================
...
induction H.                          (* Probablement beaucoup mieux que induction x. *)
```

Commandes utiles

```
Set Printing All.     (*            Désactive notations            *)
UnSet Printing All.
```

Fonction vs Relation

Typiquement :

- Expressions : retournent toujours une valeur, pas de boucle ⇒ Fonction
  - avantage : se calcule tout seul, s'extrait vers une fonction ML
  - inconvénient : Récursion structurelle
  - pas de principe d'induction
- Instructions : pas toujours défini ni unique, boucle infinies ⇒ Relation
  - avantage : principe d'induction
  - inconvénient : ne se calcule pas, pas d'extraction

Fonction vs Relation

Typiquement :

- Expressions : retournent toujours une valeur, pas de boucle ⇒ Fonction
  - avantage : se calcule tout seul, s'extrait vers une fonction ML
  - inconvénient : Récursion structurelle `Program`, `Equation`, `Function`
  - pas de principe d'induction `Equation`, `Function`
- Instructions : pas toujours défini ni unique, boucle infinies ⇒ Relation
  - avantage : principe d'induction
  - inconvénient : ne se calcule pas, pas d'extraction

Fonction vs Relation

Typiquement :

- Expressions : retournent toujours une valeur, pas de boucle ⇒ Fonction
  - avantage : se calcule tout seul, s'extrait vers une fonction ML
  - inconvénient : Récursion structurelle `Program`, `Equation`, `Function`
  - pas de principe d'induction `Equation`, `Function`
- Instructions : pas toujours défini ni unique, boucle infinies ⇒ Relation
  - avantage : principe d'induction
  - inconvénient : ne se calcule pas, pas d'extraction
    Si déterministe : fuel trick

Fonction vs Relation

Typiquement :

- Expressions : retournent toujours une valeur, pas de boucle ⇒ Fonction
  - avantage : se calcule tout seul, s'extrait vers une fonction ML
  - inconvénient : Récursion structurelle `Program`, `Equation`, `Function`
  - pas de principe d'induction `Equation`, `Function`
- Instructions : pas toujours défini ni unique, boucle infinies ⇒ Relation
  - avantage : principe d'induction
  - inconvénient : ne se calcule pas, pas d'extraction
    Si déterministe : fuel trick

Recommandation : Fonctions si possible, Relation sinon

Récursion non directement structurelle

```
Require Import NArith.

Fixpoint log10 (n : N) : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
```

Récursion non directement structurelle

```
Require Import NArith.

Fixpoint log10 (n : N) : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
```

```
Error: Recursive definition of log10 is ill-formed.
Recursive call to log10 has principal argument equal to "n/10"
instead of a subterm of "n".
```

Récursion non directement structurelle

```
Require Import NArith.

Fixpoint log10 (n : N) : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
```

Appel récursif non structurel

```
Error: Recursive definition of log10 is ill-formed.
Recursive call to log10 has principal argument equal to "n/10"
instead of a subterm of "n".
```

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence
- supposons que la définition suivante soit acceptée (Ex. B. Grégoire) :

```
Fixpoint loop (n : Z) : ? := loop n.
```

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence
- supposons que la définition suivante soit acceptée (Ex. B. Grégoire) :

```
Fixpoint loop (n : Z) : ? := loop n.
```

non contraint (démo ocaml)

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence
- supposons que la définition suivante soit acceptée (Ex. B. Grégoire) :

```
Fixpoint loop (n : Z) : False := loop n.
```

alors :

```
Check (loop 0).
  loop 0: False.                    (* Tout devient prouvable *)
```

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence
- supposons que la définition suivante soit acceptée (Ex. B. Grégoire) :

```
Fixpoint loop (n : Z) :  1=2 := loop n.
```

alors :

```
Check (loop 0).
  loop 0: 1=2.                                  (* Tout devient prouvable *)
```

Pourquoi interdit ?

- fonctions doivent terminer en Coq
- pourquoi ?
- non-terminaison $\Rightarrow$ relâchement contraintes de typage $\Rightarrow$ incohérence
- supposons que la définition suivante soit acceptée (Ex. B. Grégoire) :

```
Fixpoint loop (n : Z) : ∀ P:Prop,P := loop n.
```

alors :

```
Check (loop 0).
  loop 0: ∀ P:Prop, P.                         (* Tout devient prouvable *)
```

À propos de False

```
Inductive False : Prop := .
```

À propos de False

0 constructeur

```
Inductive False : Prop := .
```

À propos de False

0 constructeur

```
Inductive False : Prop := .
```

```
Definition absurd (P:Prop) (x:False): P:=
  match x with
  end.
```

0 branche ⇒ type non contraint, par exemple `P`

À propos de False

```
Inductive False : Prop := .
```

```
Definition absurd (P:Prop) (x:False): P:=
  match x with
  end.
```

```
H: False
==================
destruct H.                                    (*      0 cas!      *)
```

À propos de False

```
Inductive False : Prop := .
```

```
Definition absurd (P:Prop) (x:False): P:=
  match x with
  end.
```

```
Check (absurd (1=2)).                              (* False -> 1=2 *)

Axiom abs: False.
Check (absurd (1=2) abs).                          (*      1=2      *)
```

À propos de False

```
Inductive False : Prop := .
```

```
Definition absurd (P:Prop) (x:False): P:=
  match x with
  end.
```

```
Check (absurd (∀ P:Prop, P)).          (* False -> (forall P:Prop, P) *)

Axiom abs: False.
Check (absurd (∀ P:Prop, P)).          (*       forall P:Prop, P      *)
```

Récursion non directement structurelle

(`N` = `nat` en binaire.)

```
Require Import ZArith.

Fixpoint log10 (n : N) : N :=
  if Z.ltb n 10 then 0 else 1 + log10 (n / 10).
```

- refusée
- pourtant termine : pas d'incohérence
- typage pas assez intelligent
- condition de garde syntaxique

Récursion non directement structurelle

(`N` = `nat` en binaire.)

```
Require Import ZArith.

Fixpoint log10 (n : N) : N :=
  if Z.ltb n 10 then 0 else 1 + log10 (n / 10).
```

< n car Zlt_bool n 10 = fal

- refusée
- pourtant termine : pas d'incohérence
- typage pas assez intelligent
- condition de garde syntaxique
- ne sais pas que $\forall n > 0, n/10 < n$

Récursion non directement structurelle

(`N` = `nat` en binaire.)

```
Require Import ZArith.

Fixpoint log10 (n : N) : N :=
  if Z.ltb n 10 then 0 else 1 + log10 (n / 10).
```

- refusée
- pourtant termine : pas d'incohérence
- typage pas assez intelligent
- condition de garde syntaxique
- ne sais pas que $\forall n > 0, n/10 < n$
- possible de retrouver une décroissance structurelle ?

Récursion non directement structurelle

(`N` $=$ `nat` en binaire.)

```
Require Import ZArith.

Fixpoint log10 (n : N) : N :=
  if Z.ltb n 10 then 0 else 1 + log10 (n / 10).
```

- refusée
- pourtant termine : pas d'incohérence
- typage pas assez intelligent
- condition de garde syntaxique
- ne sais pas que $\forall n > 0, n/10 < n$
- possible de retrouver une décroissance structurelle ?
- oui mais à quelques ennuis près

Fuel trick

```
Fixpoint log10' (n:N) arg {struct arg} :=
  match arg with
  |     ...    => 0
  | .. arg' .. => if N.ltb n 10 then 0 else 1 + log10 (n / 10) arg'
  end.
```

Fuel trick

```
Fixpoint log10' (n:N) arg {struct arg} :=
  match arg with
  |     ...     => 0
  | .. arg' .. => if N.ltb n 10 then 0 else 1 + log10 (n / 10) arg'
  end.
```

- `arg` décroissant structurellement, ok
- quel type ?
- quelle valeur ?

Fuel trick

```
Fixpoint log10' (n:N) arg {struct arg} :=
  match arg with
  |     0      => 0
  |  S(arg')   => if N.ltb n 10 then 0 else 1 + log10 (n / 10) arg'
  end.
```

```
Eval lazy (log10 1000000 10%nat).                                (*   -> 6   *)
```

- `arg` décroissant structurellement, ok
- quel type ? nat ?
- quelle valeur ? 10, 100, 2^{100} ?

Fuel trick

```
Fixpoint log10' (n:N) arg {struct arg} :=
  match arg with
  |     0      => 0
  |  S(arg')   => if N.ltb n 10 then 0 else 1 + log10 (n / 10) arg'
  end.
```

```
Definition fuel N → nat := ?.
Definition log10 n = log10' n (fuel n).
```

- ∀n, `fuel n` ≪ assez grand ≫ pour calculer `log10' n k`.
- `fuel` de préférence calculé paresseusement.
- Démo

Problème du fuel

```
Fixpoint log10' (n:N) arg {struct arg} :=
  match arg with
  | 0 => 0
  | S arg' => if N.ltb n 10 then 0 else 1 + log10 (n / 10) arg'
  end.
```

```
Eval lazy (log10 1000000 10%nat).                    (*   -> 6    *)
Eval lazy (log10 1000000 2%nat).                     (*   -> 2!   *)
```

- Gros pb : si arg trop petit résultat faux (point-fixe non atteint)
- Preuve sur `log10` « seulement quand arg est assez grand » ?
- Inclure le calcul du arg assez grand dans la fonction
- `fuel: ∀x:N, { y | (fuel y) assez grand }`

Fuel assez grand par construction

```
Fixpoint log10' (n:N) (arg: { k | k assez grand }) {struct arg} :=
 match arg with
 |...                => 0
 |.. arg' ..               => if N.ltb n 10 then 0 else 1 + log10 (n/10) k
  end.
```

- `arg` assez grand par typage (type dépendant)
- `arg: { k | k assez grand pour calculer (log10 n) }`
- `fuel n: { k | k assez grand pour calculer (log10 n) }`

Fuel assez grand par construction

```
Fixpoint log10' (n:N) (arg: { k | k assez grand }) {struct arg} :=
 match arg with
  |(0, 0 assez grand) => 0
  |(S k, S k assez grand) => if N.ltb n 10 then 0 else 1 + log10 (n/10) k
  end.
```

- `arg` assez grand par typage (type dépendant)
- `arg: { k | k assez grand pour calculer (log10 n) }`
- `fuel n: { k | k assez grand pour calculer (log10 n) }`

Fuel assez grand par construction

- Loi de point fixe : itérer plus de *p* fois la fonctionnelle (`log10_F`) ne change pas le résultat :

```
fuel n:
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)
            → ∀ def:N → N, iter (N → N) k log10_F def n = v }
```

Fuel assez grand par construction

- Loi de point fixe : itérer plus de p fois la fonctionnelle (`log10_F`) ne change pas le résultat :

```
fuel n:
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)
            → ∀ def:N → N, iter (N → N) k log10_F def n = v }
```

- `arg` contient la valeur de retour (`v`) !

Fuel assez grand par construction

- Loi de point fixe : itérer plus de p fois la fonctionnelle (`log10_F`) ne change pas le résultat :

```
fuel n:
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)
            → ∀ def:N → N, iter (N → N) k log10_F def n = v }
```

- `arg` contient la valeur de retour (`v`) !
- changement de point de vue :

Fuel assez grand par construction

- Loi de point fixe : itérer plus de p fois la fonctionnelle (`log10_F`) ne change pas le résultat :

```
fuel n:
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)
           → ∀ def:N → N, iter (N → N) k log10_F def n = v }
```

- `arg` contient la valeur de retour (`v`) !
- changement de point de vue :
  - `fuel` est une fonction qui retourne une paire :
    (résultat , preuve qu'on a itéré suffisamment)
  - `log10` : appel de `fuel` + π_1

Outil Function

```
Function log10 (n : N) { wf Nlt n } : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
Proof.
  - ... (* Preuve de la décroissance des appels récursif selon Nlt *)
  - ... (* Preuve que Nlt est << bien fondée >>. *)
Defined.
```

- construit automatiquement :

```
log10_F = fun (log10: N → N)(n:N) => if n<?10 then 0 else 1+log10(n/10)
log10_terminate: ∀ n:N,
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)%nat
              → ∀ def:N → N, iter (N → N) k log10_F def n = v }
log10 = (let (v, _) := log10_terminate x in v) : N → N
log10_equation: ∀ n:N, log10 n = (if n<?10 then 0 else 1 + log10 (n/10))
```

Outil Function

```
Function log10 (n : N) { wf Nlt n } : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
Proof.
  - ... (* Preuve de la décroissance des appels récursif selon Nlt *)
  - ... (* Preuve que Nlt est << bien fondée >>. *)
Defined.
```

- construit automatiquement :

```
log10_F = fun (log10: N → N)(n:N) => if n<?10 then 0 else 1+log10(n/10)
log10_terminate: ∀ n:N,
  { v : N | ∃ p:nat, ∀ k:nat, (p < k)%nat
      fuel    → ∀ def:N → N, iter (N → N) k log10_F def n = v }
log10 = (let (v, _) := log10_terminate x in v) : N → N
log10_equation: ∀ n:N, log10 n = (if n<?10 then 0 else 1 + log10 (n/10))
```

Outil Function

```
Function log10 (n : N) { wf Nlt n } : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
Proof.
  - ... (* Preuve de la décroissance des appels récursif selon Nlt *)
  - ... (* Preuve que Nlt est << bien fondée >>. *)
Defined.
```

- construit automatiquement :

```
log10_ind: ∀ P: N → N → Prop,
             (∀ n:N, (n <? 10) = true → P n 0) →
             (∀ n:N, (n <? 10) = false → P (n/10) (log10 (n/10))
                → P n (1 + log10 (n/10))) →
           ∀ n:N, P n (log10 n)
```

- induction suivant le schéma de la fonction (`functional induction`)
- Démo

Outil Program Fixpoint

```
Program Fixpoint log10 (n : N) { wf Nlt n } : N :=
  if N.ltb n 10 then 0
  else 1 + log10 (n / 10).
Next Obligation.
  ... (* Preuve de la décroissance des appels récursif selon Nlt *)
Defined.
Next Obligation.
  ... (* Preuve que Nlt est "bien fondée". *)
Defined.
```

- construit automatiquement :

```
log10' = Fix_sub N ... log10'_obligation₂ ... log10'_obligation₁
  : N → N
Fix_sub = ...
 : ∀ (A:Type) (<_R:A → A → Prop), well_founded <_R → ∀ P:A → Type,
       (∀ x : A, (∀ y : {y' : A | y' <_R x}, P (proj1_sig y)) → P x)
    → ∀ x : A, P x.
```

Ordre bien fondés

```
well_founded = fun (A : Type) (R : A → A → Prop) => ∀ a : A, Acc R a

Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
    Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

Récursion non directement structurelle

```
Require Import FunInd Recdef List Arith. Import Nat ListNotations.

Definition slen (p:list nat * list nat):= length(fst p) + length(snd p).

Function Merge (p:list nat * list nat) {measure slen p}: list nat :=
  match p with
   | (nil, l2) => l2
   | (l1, nil) => l1
   | ((x1::l1') as l1, (x2::l2') as l2) =>
      if leb x1 x2 then x1::Merge (l1',l2)
      else x2::Merge (l1,l2')
  end.
Proof.
- intros. auto with arith.
- intros. unfold slen; auto with arith.
Defined.          (* Pour garder la définition dépliable (≠ lemmes)  *)

Compute Merge (2::3::5::7::nil, 3::4::10::nil).
```

Récursion non directement structurelle

```
Require Import Coq.Program.Wf List Arith.
Import Nat ListNotations.

Definition slen (p:list nat * list nat):= length(fst p) + length(snd p).

Program Fixpoint Merge (p:list nat * list nat) {measure (slen p)}: list na
  match p with
  | (nil, l2) => l2
  | (l1, nil) => l1
  | ((x1::l1') as l1, (x2::l2') as l2) =>
      if leb x1 x2 then x1::Merge (l1',l2)
      else x2::Merge (l1,l2')
  end.
Next Obligation.
- intros. unfold slen; auto with arith.
Qed.

Compute Merge (2::3::5::7::nil, 3::4::10::nil).
```

Function vs Program Fixpoint

- `Function` produit plus d'outils :
  - `functional induction`
  - `functional inversion`
  - utile même pour les fonctions récursives structurelles
- `Program Fixpoint` supporte :
  - les types dépendants

```
Program Fixpoint log10' (n : N) { wf Nlt n }
   : {x:N | power 10 (x-1) < n <= power 10 x} :=
  match  N.ltb n 10 with
  | true => 0
  | false => 1 + log10' (n / 10)
  end.
```

  - les notations (Régression récente de `Function`)

FIN

Fin

Preuve de programmes fonctionnels

```
let rec f x y = if x < y then 0 else 1 + f (x-y) y
```

```
Function f x y = if x < y || y ?= 0 then 0 else 1 + f (x-y) y.

Lemma ∀ x y:N,
  x >= 0 → y > 0 → (f x y) * y <= x /\ (1 + f x y) * y > y.
Proof. ...
Defined.

Lemma foo:∀ x y, x>=0 → y>0 → (f x y)*y <= x /\ (1 + f x y)*y > x.
Proof.
  intros x y. functional induction f x y;intros. ...
```

- preuve sur une fonction mathématique = Maths
- spécification d'une fonction : précondition + relation entre paramètres et résultat :

$$Precond(x_1, ...x_n) \rightarrow Correct(x_1, ...x_n, f(x_1, ...x_n))$$

Quid des programme impératifs ?

- programme avec effet de bord
- objet mathématique moins « propre »
- programme = transformateur d'état
- propriété d'un programme ?

Spécification

- P transforme un état q_1 t.q. $Precond(q_1)$ en un état q_2 t.q. $Postcond(q_2)$
- P = transformateur de propriété

- Spécification : $\mathcal{S}(q_1, q_2) = P(q_1) \Rightarrow Q(q_2)$.
- P : Pré-condition, Q : Post-condition.

Spécification

- P transforme un état q_1 t.q. $Precond(q_1)$ en un état q_2 t.q. $Postcond(q_2)$
- P = transformateur de propriété

- Spécification : $\mathcal{S}(q_1, q_2) = P(q_1) \Rightarrow Q(q_2)$.
- P : Pré-condition, Q : Post-condition.
- Triplet de Hoare :

$$\{P\}programme\{Q\}$$

Exemple de triplet $\{P\}prog\{Q\}$

```
P:
 a := 0;
 b := x;
 while (b >= y) do
    b := b - y;
    a := a + 1
 done
```

Spécifications possibles :

- $\{x \geq 0 \ \wedge \ y > 0\}\ P\ \{a * y \leq x \ \wedge \ (1 + a) * y > x\}$
- $\{x >= 0 \ \wedge \ y > 0\}\ P\ \{a * y + b = x \ \wedge \ 0 \leq b < y > x\}$

Triplet correct : 2 versions

- Triplet $\{P\}prog\{Q\}$ (partiellement) correct ssi :

$$\forall q_1, P(q_1) \rightarrow \quad (\forall q_2, \langle q_1, prog\rangle \rightsquigarrow q_2 \rightarrow Q(q_2))$$

- Triplet $\langle P\rangle prog\langle Q\rangle$ (totalement) correct ssi :

$$\forall q_1, P(q_1) \rightarrow \quad \begin{array}{ll} & (\langle q_1, prog\rangle \rightsquigarrow \text{termine toujours}) \\ \wedge & (\forall q_2, \langle q_1, prog\rangle \rightsquigarrow q_2 \rightarrow Q(q_2)) \end{array}$$

- Si sémantique déterministe : Correction totale reformulable :

$$\forall q_1, P(q_1) \rightarrow \quad (\exists q_2, \langle q_1, prog\rangle \rightsquigarrow q_2 \rightarrow Q(q_2))$$

Règles AFF et SEQ

$$\textsc{Aff1}\ \frac{}{\{P[x \leftarrow E]\}\ x := E\ \{P\}}$$

$$\textsc{Aff}\ \frac{}{\{P\} \quad x := E \quad \{P[x \leftarrow x_0] \wedge x = E[x \leftarrow x_0]\}}$$

$$\textsc{Seq}\ \frac{\{P\}C_1\{Q\} \qquad \{Q\}C_2\{R\}}{\{P\}C_1\ ;C_2\{R\}}$$

Règles COND et CONSEQ

$$\textsc{Conseq}\ \frac{P \Rightarrow P' \qquad \{P'\}C\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

$$\textsc{Cond}\ \frac{\{P \wedge B\}\ I_1\ \{Q\} \qquad \{P \wedge \neg B\}\ I_2\ \{Q\}}{\{P\}\ \mathit{if}\ \ B\ \ \mathit{then}\ I_1\ \mathit{else}\ I_2\ \{Q\}}$$

Règles du while

$$\textsc{While}\ \frac{\{P \wedge B\}\ C\ \{P\}}{\{P\}\ \mathit{while}\ \ B\ \mathit{do}\ \ C\ \mathit{done}\ \{P \wedge \neg B\}}$$

$$\textsc{WhileT}\ \frac{\langle P\ \wedge\ B\ \wedge\ (E = n)\rangle\ C\ \langle\ P\ \wedge\ E\ <\ n\ \wedge\ E\ \geq\ 0\ \rangle}{\langle P\rangle\ \mathit{while}\ B\ \mathit{do}\ C\ \mathit{done}\ \langle P \wedge \neg B\rangle}$$

TP

- `http://cedric.cnam.fr/~courtiep/downloads/plf-svp.tgz`
- `Equiv.v`
- `Hoare.v`