Le Coq au Pauillac et aux omégas

(Aprieta el bastón con las dos manos, se yergue un tanto, casi con entusiasmo) ¡Caramba! Claro... los números transfinitos, Kantor... *[Jorge Luis Borges]*

Nous essayons d'orienter toute la production animale vers des produits enrichis en oméga-3 par l'alimentation: nous proposons déjà du poulet, de la bleue des prés et du porc. Bientôt la charcuterie va suivre. *[Le Responsable qualité d'une chaîne d'hypermarchés (La Dernière Heure, 2005)]*

auto. *[Moi]*

Proof completed *[Coq]*

Qed *[Moi]*

Pauillac,

samedi 28 Janvier,
3 heures 30 de l'après-midi
. . .

Mission impossible

“ Vous devrez présenter Coq en trois heures, sans perdre les débutants ni ennuyer les personnes ayant déjà une expérience minimum.
Bien entendu, en cas d’échec, la Direction des Journées déclinera toute responsabilité ”

Contenu

On privilégiera le point de vue de l'utilisateur, en se concentrant sur quelques thèmes:

- relations entre calcul et inférence
- preuves de terminaison
- Coq et les mathématiques

Certains aspects — fort intéressants — ne pourront être traités:

- Preuves de programmes impératifs: `Why`, `Krakatoa`, `Caduceus`
- Structures de données potentiellement infinies,
- Systèmes de modules,
- "Art of tactic programming".

A propos de Goodstein

Les suites de Goodstein et des problèmes similaires (batailles d'Hydre) présentent les caractéristiques suivantes:

- Propriétés faciles à énoncer, parfois assez contre-intuitives:
- Les preuves de ces propriétés mêlent inférences et calculs

Ces caractéristiques en font un bon thème d'introduction aux assistants de preuve, et en particulier à Coq.

La suite G_{42}

$$\begin{aligned} 42 &= 2^5 + 2^3 + 2 \\ &= 2^{(2^2+1)} + 2^{2+1} + 2 \end{aligned}$$

La suite G_{42}

$$
\begin{aligned}
42 &= 2^5 + 2^3 + 2 \\
&= 2^{(2^2+1)} + 2^{2+1} + 2 \\
\bullet \quad &= 3^{(3^3+1)} + 3^{3+1} + 3 - 1 \\
&= 3^{(3^3+1)} + 3^{3+1} + 2
\end{aligned}
$$

La suite G_{42}

$$
\begin{aligned}
42 &= 2^5 + 2^3 + 2 \\
&= 2^{(2^2+1)} + 2^{2+1} + 2 \\
\bullet \quad &\quad 3^{(3^3+1)} + 3^{3+1} + 3 - 1 \\
&= 3^{(3^3+1)} + 3^{3+1} + 2 \\
\bullet \quad &\quad 4^{(4^4+1)} + 4^{4+1} + 1
\end{aligned}
$$

La suite G_{42}

$$
\begin{aligned}
42 &= 2^5 + 2^3 + 2 \\
&= 2^{(2^2+1)} + 2^{2+1} + 2 \\
\bullet \quad &\quad 3^{(3^3+1)} + 3^{3+1} + 3 - 1 \\
&= 3^{(3^3+1)} + 3^{3+1} + 2 \\
\bullet \quad &\quad 4^{(4^4+1)} + 4^{4+1} + 1 \\
\bullet \quad &\quad 5^{(5^5+1)} + 5^{5+1}
\end{aligned}
$$

- $6^{(6^6+1)} + 6^{6+1} - 1$

- $$\begin{aligned} & 6^{(6^6+1)} + 6^{6+1} - 1 \\ = \quad & 6^{(6^6+1)} + 6^6 \times 5 + \\ & 6^5 \times 5 + 6^4 \times 5 + \cdots + 6 \times 5 + 5 \end{aligned}$$

- $6^{(6^6+1)} + 6^{6+1} - 1$

  $= 6^{(6^6+1)} + 6^6 \times 5 +$

  $6^5 \times 5 + 6^4 \times 5 + \cdots + 6 \times 5 + 5$

  $\ldots$

- $11^{(11^{11}+1)} + 11^{11} \times 5 +$

  $11^5 \times 5 + 11^4 \times 5 + \cdots + 11 \times 5$

  $\ldots$???

Un exemple plus simple : G_4

Si l'on part de 4 en base 2, la suite est plus simple à étudier:

$$4 \quad = \quad 2^2$$

Un exemple plus simple : G_4

Si l'on part de 4 en base 2, la suite est plus simple à étudier:

$$\begin{array}{lll} 4 & = & 2^2 \\ \bullet & & 3^2 \times 2 + 3 \times 2 + 2 \end{array}$$

Un exemple plus simple : G_4

Si l'on part de 4 en base 2, la suite est plus simple à étudier:

$$
\begin{array}{rcl}
4 & = & 2^2 \\
\bullet & & 3^2 \times 2 + 3 \times 2 + 2 \\
\bullet & & 4^2 \times 2 + 4 \times 2 + 1
\end{array}
$$

Un exemple plus simple : G_4

Si l'on part de 4 en base 2, la suite est plus simple à étudier:

$$
\begin{array}{rcl}
4 & = & 2^2 \\
\bullet & & 3^2 \times 2 + 3 \times 2 + 2 \\
\bullet & & 4^2 \times 2 + 4 \times 2 + 1 \\
\bullet & & 5^2 \times 2 + 5 \times 2 + 0
\end{array}
$$

Un exemple plus simple : G_4

Si l'on part de 4 en base 2, la suite est plus simple à étudier:

$$
\begin{aligned}
4 &= 2^2 \\
&\bullet \quad 3^2 \times 2 + 3 \times 2 + 2 \\
&\bullet \quad 4^2 \times 2 + 4 \times 2 + 1 \\
&\bullet \quad 5^2 \times 2 + 5 \times 2 + 0 \\
&\bullet \quad 6^2 \times 2 + 6 \times 1 + 5 \\
&\ldots
\end{aligned}
$$

Cette suite termine et converge vers 0 (ordre lexicographique sur les coefficients).

Un petit programme en *OCaml*

```
let g4_length =
 (* length of the sequence starting from
    b^2 x + b y + z  *)
 let rec aux  b x y z =
       if z > 0
       then aux (b + 1)  x y (z - 1)
       else if y > 0
       then aux (b + 1)  x (y - 1) b
       else if x > 0
       then aux (b + 1) (x - 1) b b
       else b - 1
 in aux 3 2 2 2;;
```

Un petit programme en *OCaml*

```
let g4_length =
 (* length of the sequence starting from
    b^2 x + b y + z  *)
 let rec aux  b x y z =
       if z > 0
       then aux (b + 1)  x y (z - 1)
       else if y > 0
       then aux (b + 1)  x (y - 1) b
       else if x > 0
       then aux (b + 1) (x - 1) b b
       else b - 1
 in aux 3 2 2 2;;
```

Attente . . .

Un petit programme en *OCaml*

```
let g4_length =
 (* length of the sequence starting from
    b^2 x + b y + z  *)
 let rec aux  b x y z =
       if z > 0
       then aux (b + 1)  x y (z - 1)
       else if y > 0
       then aux (b + 1)  x (y - 1) b
       else if x > 0
       then aux (b + 1) (x - 1) b b
       else b - 1
 in aux 3 2 2 2;;

^C
```

Calculer en Coq

Pour calculer la longueur de G_4, nous devrons recourir à un peu de raisonnement.
Avant de procéder à cette évaluation, nous pouvons présenter les quelques traits qui permettent de combiner calculs et inférences dans un même développement en Coq.
Commençons par quelques exemples simples.

Mon premier Théorème

```
Theorem my_first_theorem : 2 + 2 = 2 * 2.
Proof.
1 subgoal

  ============================
   2 + 2 = 2 * 2

 trivial.
Proof completed.
Qed.
my_first_theorem is defined
```

Les composants d'une preuve

Les entiers naturels sont définis comme un type de donnée inductif, à l'aide de deux *constructeurs*: O (0) et S (successeur).

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

O et S sont les constructeurs du type inductif nat.
Set est le type (*sorte*) des spécifications.

Les composants d'une preuve

Les entiers naturels sont définis comme un type de donnée inductif, à l'aide de deux *constructeurs*: O (0) et S (successeur).

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

Note : 0 est une abréviation de O, 2 une abréviation de S (S O), etc.

Sortes et univers

En Coq, tout terme bien formé a un type, et tout type est lui même un terme du *Calcul des Constructions Inductives*: Le type d'un type est appelé une *sorte*.

```
Set : Type(0)
Type(i) : Type(j)  (i < j)

nat : Set
nat : Type(i)
```

Jugements de typage

Types fonctionnels:

```
plus, mult : nat → nat → nat
nat → nat → nat : Set
nat → nat → nat : Type
```

Applications:

```
plus 2 : nat → nat
2 + 2 : nat [plus 2 2  i.e (plus 2) 2]
2 * 2 : nat
```

Abstractions:

```
fun (n:nat) ⇒ n + n : nat → nat
```

Produit dépendant

```
eq: ∀ A : Type, A → A → Prop

(* cf OCaML : (=) : 'a -> 'a -> bool *)

2+2 = 2*2 : Prop
(eq nat (plus 2 2) (mult 2 2))

∀ b, negb (negb b) = b : Prop
forall b:bool, eq bool (negb (negb b)) b
```

filtrage et récursion structurelle

```
Fixpoint plus (n m:nat) {struct n} : nat :=
match n with
| O ⇒ m
| S p ⇒ S (p +  m)
end
where "n + m" := (plus n m) : nat_scope.
```

$$\begin{aligned} 0 + t &\rightarrow_{\beta\delta\iota} t \\ (S\ t) + t' &\rightarrow_{\beta\delta\iota} S(t + t') \end{aligned}$$

Terme de preuve

```
refl_equal: ∀ (A:Type)(a:A), a=a
```

```
refl_equal nat 4 : 4 = 4
```

$\equiv$ *2+2 = 2*2 (règle de conversion)*

On ajoute alors à l'environnement une constante `my_first_theorem` de type 2+2=2*2, et de valeur refl_equal nat 4.

Deux preuves d'existence

```
Lemma ex1 : ∃ n:nat, n+n = n*n.
Proof.
  exists 0;trivial.
Qed.

Lemma ex2 : ∃ n:nat, n+n = n*n.
Proof.
  exists 2;trivial.
Qed.
```

Les deux termes de preuve

```
Print ex1.
ex_intro (fun n : nat ⇒ n + n = n * n) 0
         (mult_n_O 0)
     : ∃ n : nat, n + n = n * n
Print ex2.
ex_intro (fun n : nat ⇒ n + n = n * n) 2
         (refl_equal (2 * 2))
     : ∃ n : nat, n + n = n * n
About ex_intro.
ex_intro :
 ∀ (A : Type) (P : A → Prop) (x : A), P x → ex P
Argument A is implicit
```

Prouver sans tactiques ?

```
Definition ex3 : ∃ n:nat, n+n = n*n :=
  exists_intro (fun p : nat ⇒ p+p = p*p)
               2 (refl_equal 4).

Opaque ex3.
```

Réduire ou pas

```
Lemma zero_plus : ∀ n:nat, 0+n=n.
Proof.
 trivial.
Qed.
```

Le terme de preuve est étonnamment simple:

```
Print zero_plus.
fun n:nat ⇒ refl_equal nat  n
```

Vérification de type

$$\dfrac{\dfrac{\dfrac{\left\{\begin{array}{l}\texttt{n:nat}\\ \texttt{refl\_equal:}\forall\texttt{(A:Type) (a:A), a=a}\\ \texttt{nat:Type}\end{array}\right.\vdash \texttt{n:nat}}{\texttt{n:nat} \vdash \texttt{refl\_equal nat n} : n = n}\,\mathit{app}}{\texttt{n:nat} \vdash \texttt{refl\_equal nat n} : 0 + n = n}\,\mathit{conv}}{\texttt{fun n:nat}\Rightarrow\texttt{refl\_equal nat n} : \forall n,\ 0 + n = n}\,\mathit{abs}$$

En revanche ...

```
Lemma plus_zero : ∀ n:nat, n+0=n.
Proof.
  induction n.
```

La tactique induction n utilise le théorème suivant (engendré à la compilation du type nat):

```
nat_ind : ∀  P : nat → Prop,
       P 0 →
       (∀ n : nat, P n → P (S n)) →
       ∀ n : nat, P n
```

La conclusion P n, confrontée au sous-but n+0=n construit l'instantiation $P = (\lambda p.\, p + 0 = p)$.

L'application de nat_ind engendre deux sous-buts correspondant respectivement à $P\ 0$ et $\forall n, P\ n \Rightarrow P(S\ n)$

```
============================
 0 + 0 = 0
```

```
trivial.
```

```
n : nat
IHn : n + 0 = n
============================
 (S n) + 0 = S n
```

```
  n : nat
  IHn : n + 0 = n
  ============================
   (S n) + 0 = S n

simpl.
  n : nat
  IHn : n + 0 = n
  ============================
   S (n + 0) = S n
rewrite IHn;trivial.
Qed.
```

Les tactiques induction, rewrite, simpl et trivial nous ont évité d'expliciter la définition suivante:

```
Fixpoint plus_zero' (n:nat) : n+0=n :=
  match n return n+0=n with
     0 ⇒ refl_equal 0
   | S p ⇒ eq_ind_r (fun n1 : nat ⇒ S n1 = S p)
                    (refl_equal (S p)) (plus_zero' p)
  end.

Opaque plus_zero'.
```

Pour comprendre la définition précédente, on a recours aux jugements de typage:
Le theorème eq_ind_r permet d'éliminer l'égalité (en utilisant la symétrie).

```
eq_ind_r : ∀ (A : Type) (x : A) (P : A → Prop),
       P x → ∀ y : A, y = x → P y
```

Dans un contexte où plus_zero' p a pour type p+0 = p, le terme :

```
eq_ind_r  (fun n1 : nat ⇒ S n1 = S p)
          (refl_equal (S p)) (plus_zero' p)
```

a pour type S (p + 0) = S p, convertible avec S p + 0 = S p.

En résumé

Coq est basé sur un lambda-calcul typé, le *Calcul des Constructions Inductives*, et possède les caractéristiques suivantes:

- Des règles de typage assurent la bonne formation des termes
- Des règles de conversion définissent une congruence entre termes; le calcul associé est fortement normalisant et confluent,
- Une proposition est un terme de type `Prop`

En résumé

- La preuve d'une proposition P est un terme de type P
- Une spécification (sens très large) est un terme de type `Set`,
- Une réalisation d'une spécification A est un terme de type A.
- La construction de termes d'un type donné est facilitée par des `tactiques`.

Calcul de $|G_4|$ en Coq

```
Record item :Set :=
 quad {  base: nat ;
         coeff2 :nat ; (* coefficient of b^2 *)
         coeff1 :nat ; (* coefficient of b *)
         coeff0 :nat   (* coefficient of b^0  *)
}.

Check (quad 3 2 2 2).
quad 3 2 2 2 : item
```

```
Inductive Rg: item → item →Prop :=
| Rg0 : ∀ b i j k, Rg (quad b i j (S k))
                     (quad (S b) i j k)
| Rg1 : ∀ b i j,   Rg (quad b i (S j) 0)
                     (quad (S b) i j b)
| Rg2 : ∀ b i,     Rg (quad b (S i) 0 0)
                     (quad (S b) i b b).

Hint Constructors Rg.
```

```
Cf Prolog:
rg(quad(B,I,J,s(K)), quad(s(B),I,J,K)).
rg(quad(B,I,s(J),0), quad(s(B),I,J,B)).
rg(quad(B,s(I),0,0), quad(s(B),I,B,B)).
```

La bibliothèque Relations nous permet de définir la fermeture transitive et réflexive de R_g.

```
Definition Rgstar := clos_refl_trans _ Rg.

Definition reachable (q:item) :=
   Rgstar (quad 3 2 2 2) q.
```

But : construire un nombre naturel N et une preuve de la proposition reachable (quad N 0 0 0).

Remarque: on aurait pu utiliser *OCaml* pour étudier des *petits* morceaux de G_4.

```
let next q =  match q with
|Quad(_,0,0,0) → None
|Quad(b,x,0,0) → Some(Quad(b+1,x-1,b,b))
|Quad(b,x,y,0) → Some(Quad(b+1,x,y-1,b))
|Quad(b,x,y,z) → Some(Quad(b+1,x,y,z-1));;
```

```
next : item → item option
```

```
let rec nexts n q =
  match n with 0 → Some q
             | n → (match (next q) with
                          None → None
                        | Some q' → nexts (n-1) q');;

nexts : int → item → item option

let nth_item n = nexts n (Quad(3,2,2,2));;

nth_item : int → item option
```

Le choix de *Coq* conduit à des programmes plus complexes, mais dont la correction peut être assurée. On évite également l'hétérogénéité des outils.

```
Inductive final : item → Prop :=
 final_intro :∀ b , final (quad b 0 0 0).

Hint Constructors final.
```

Spécification forte

```
Definition next (q:item) : {q' : item | Rg q q'} +
                           {final q}.
 intros q; case q ; intros b x y z.
 case z.
 ...
Defined.
```

Valeurs possibles pour next q:

- `inleft` q π avec π : `Rg` q q'
- `inright` π' avec π' : `final` q

Cf. les valeurs `Some` q et `None` de type : `item option` en *OCaml*.

Extraction vers Ocaml

```
let next = function
 { base = b; coeff2 = x; coeff1 = y; coeff0 = z } →
 (match z with
  | O →
   (match y with
    | O →
       (match x with
        | O → Inright
        | S x' → Inleft { base = (S b); coeff2 = x';
                          coeff1 = b; coeff0 = b })
        | S y' → Inleft { base = (S b); coeff2 = x;
                          coeff1 = y'; coeff0 = b })
        | S z' → Inleft { base = (S b); coeff2 = x;
                          coeff1 = y;  coeff0 = z' })
```

La fonction `nexts` se programme (presque) comme en *OCaml*:

```
Fixpoint nexts  (n:nat)(q:item) {struct n}
      : option item :=
  match n with 0 ⇒ Some q
               | S p ⇒
    (match (next q) with
     | inleft (exist q' _) ⇒  nexts p q'
     | inright  _ ⇒ None
    end)
  end.
```

```
nexts : nat → item → option item
(Spécification faible)
```

On peut prouver la correction de `nexts`:

```
Lemma nexts_ok : ∀ n q q', nexts n q  = Some q' →
                                Rgstar q q'.
Proof.
 induction n; simpl; auto.
 injection 1; destruct 1;constructor 2.
 intros q q'; case (next  q).
 destruct s.
 intro.
 constructor 3 with x;auto.
```

Utilisation de l'hypothèse de récurrence

```
 n : nat
IHn : ∀ q q' : item, nexts n q = Some q' → Rgstar q q'
q : item
q' : item
x : item
r : Rg q x
H : nexts n x = Some q'
============================
 Rgstar x q'
```

```
apply IHn;auto.
```

```
n : nat
q : item
q' : item
============================
 final q → None = Some q' → Rgstar q q'
```

```
discriminate 2.
Qed.
```

Exercice

On prouvera la réciproque de nexts_ok.

```
Lemma nexts_plus : ∀ n p q ,
  nexts (n+p) q = match nexts n q with
                  | Some q' ⇒ nexts p q'
                  | None ⇒ None
                  end.

...

Lemma nexts_ok_R : ∀ q q', Rgstar q q' ->
                        ∃ n, nexts n q  = Some q'.
Proof.
 intros q q' H; elim H
(* récurrence sur l'hypothese Rgstar q q' *)
```

Un peu de calcul

```
Definition nth_item n := nexts n (quad 3 2 2 2).

Eval compute in nth_item 0.
  = Some (quad 3 2 2 2)
     : option item

Eval compute in nth_item 2.
    = Some (quad 5 2 2 0)
     : option item
```

A la recherche de régularités

Par tâtonnements successifs, nous obtenons (toujours avec Eval compute) les résultats suivants:

```
nth_item 3 ⇝  Some (quad 6 2 1 5)
nth_item 8 ⇝  Some (quad 11 2 1 0)
nth_item 20 ⇝ Some (quad 23 2 0 0)
nth_item 21 ⇝ Some (quad 24 1 23 23)
nth_item 44 ⇝ Some (quad 47 1 23 0)
nth_item 92 ⇝ Some (quad 95 1 22 0)
nth_item 188 ⇝ Some (quad 191 1 21 0)
```

A la recherche de régularités

Par tâtonnements successifs, nous obtenons (toujours avec Eval compute) les résultats suivants:

```
nth_item 3 ⇝  Some (quad 6 2 1 5)
nth_item 8 ⇝  Some (quad 11 2 1 0)
nth_item 20 ⇝ Some (quad 23 2 0 0)
nth_item 21 ⇝ Some (quad 24 1 23 23)
nth_item 44 ⇝ Some (quad 47 1 23 0)
nth_item 92 ⇝ Some (quad 95 1 22 0)
nth_item 188 ⇝ Some (quad 191 1 21 0)
```

$11 = 3 * 2^2 - 1$
$23 = 3 * 2^3 - 1$
$47 = 3 * 2^4 - 1$
$95 = 3 * 2^5 - 1$
$191 = 3 * 2^6 - 1$

Définissons une fonction auxiliaire

```
Fixpoint power (b:nat)(n:nat){struct n}:nat :=
 match n with
 | 0 ⇒ 1
 | S p ⇒ b * b ^ p
 end
where "n ^ p" := (power n p):nat_scope.

Definition f n := pred (3 * 2^n).
```

Une preuve (un peu paresseuse)

```
Lemma f_Sn : ∀ n,  f (S n) = S (2*(f n)).
Proof.
 induction n ;simpl.
 trivial.
 unfold f;simpl.
 unfold f at 1 in IHn.
 simpl in IHn.
 omega.
Qed.
```

Quelques lemmes (récurrences simples)

Les lemmes suivants s'obtiennent facilement (nous avons supprimé les quantifications universelles pour plus de lisibilité).

$$\frac{\mathrm{reachable}(b, i, j, k)}{\mathrm{reachable}(k + b, i, j, 0)}\ L2$$

$$\frac{\mathrm{reachable}(\mathrm{f}(b), i, j + 1, 0)}{\mathrm{reachable}(\mathrm{f}(b + 1), i, j, 0)}\ L4$$

$$\frac{\mathrm{reachable}(\mathrm{f}(b), i, k + j, 0)}{\mathrm{reachable}(\mathrm{f}(k + b), i, j, 0)}\ L5$$

Retour aux calculs

Quadruplets accessibles:

```
Lemma F3 : reachable (quad (f 3) 2 0 0).
Proof.
 compute.
 apply nexts_ok with 20; trivial.
Qed.

Print F3.
F3 =
nexts_ok 20 (quad 3 2 2 2) (quad 23 2 0 0)
  (refl_equal (Some (quad 23 2 0 0)))
     : reachable (quad (f 3) 2 0 0)
```

Retour aux calculs

Quadruplets accessibles:

$$(f(3)+1, 1, f(3), f(3)) \quad \text{(F3 et définition de } R_g\text{)}$$
$$(f(4), 1, f(3), 0) \quad \text{(f\_Sn et L2)}$$
$$(f(27), 1, 0, 0) \quad \text{(L5) et } 27 = 4 + f(3)$$
$$(f(27)+1, 0, f(27), f(27)) \quad \text{(Définition de } R_g\text{)}$$
$$(f(28), 0, f(27), 0) \quad \text{(f\_Sn et L2)}$$
$$(f(f(27)+28), 0, 0, 0) \quad \text{(L5)}$$

Evaluation de la longueur de G4

```
Require Import ZArith.

Open Scope Z_scope.
Definition f_Z n := 3*(Zpower 2 n)-1.

Eval compute in  (f_Z 27 + 28).
    = 402653211
      : Z
```

La suite G_4 a donc $3 \times 2^{402653211} - 1$ éléments! On comprend pourquoi, malgré la rapidité d'*OCaML*, l'évaluation de `g4_length` pouvait durer encore longtemps.

récapitulatif provisoire

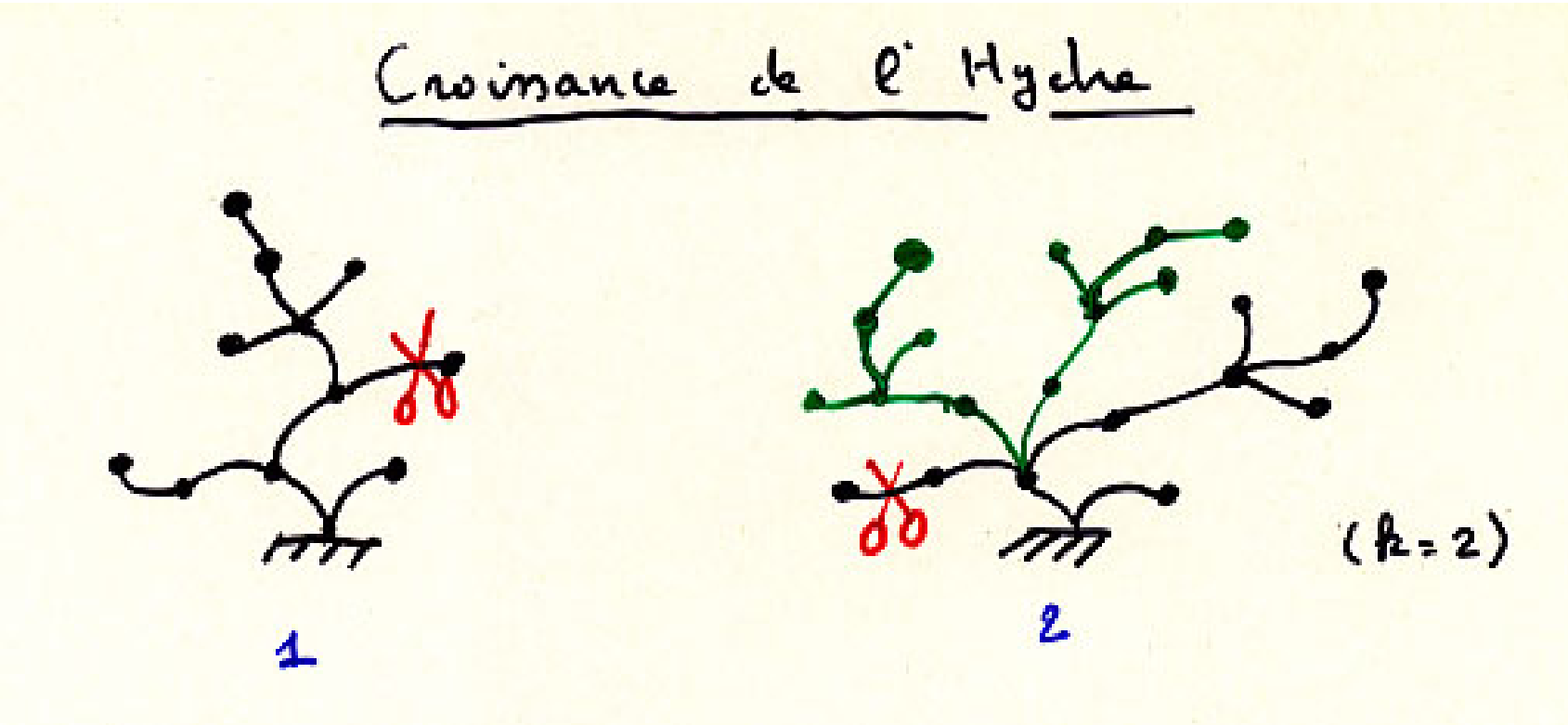
- Capacités de calcul (récursion structurelle + ordre supérieur),
- Preuve et application de théorèmes,
- Système de types expressif (types dépendants, inductifs, etc.)
- Interface déduction/calcul.

Problèmes de terminaison

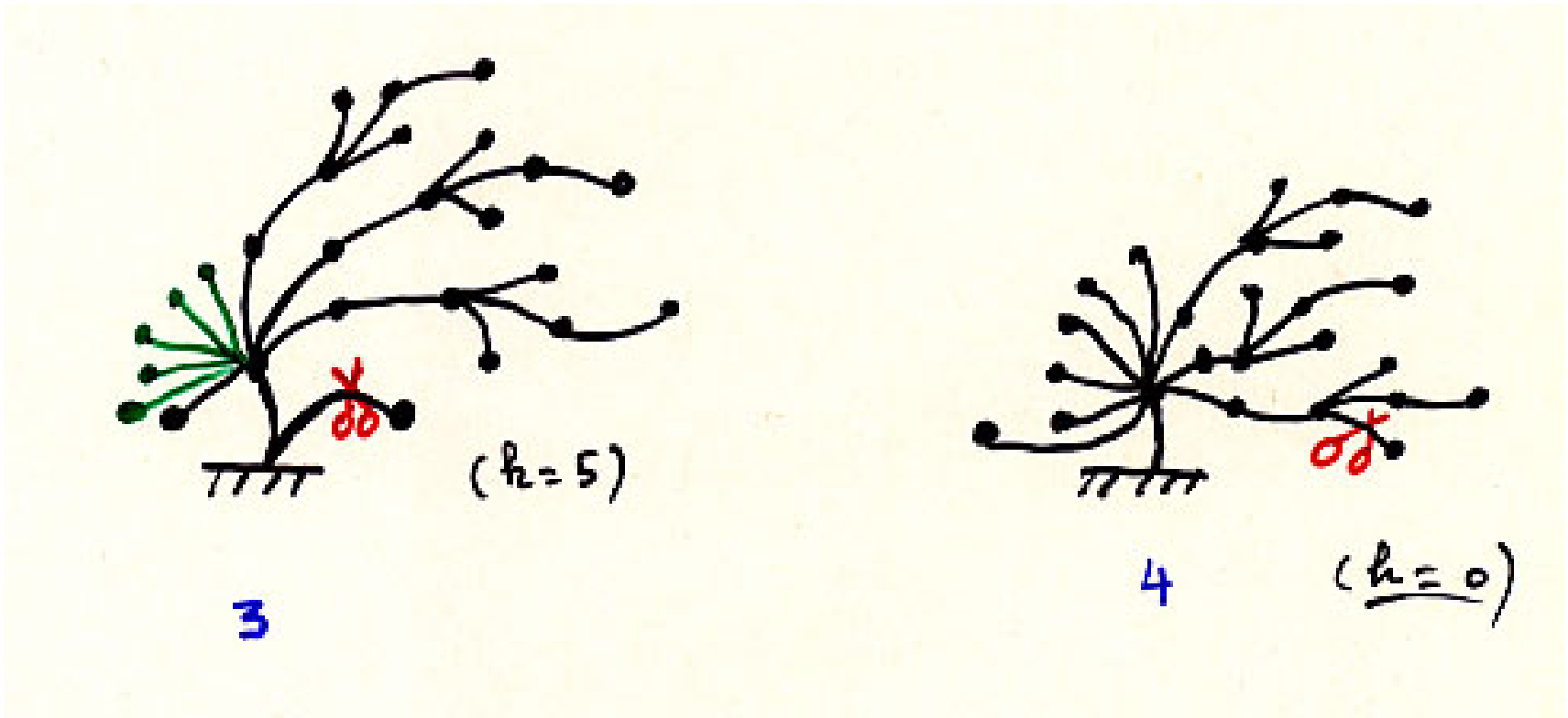
Nous illustrons quelques techniques de preuve à l'aide de quelques problèmes non triviaux.

- Batailles d'Hydre
- Terminaison de toutes les suites de Goodstein,
- L'ordinal ϵ_0

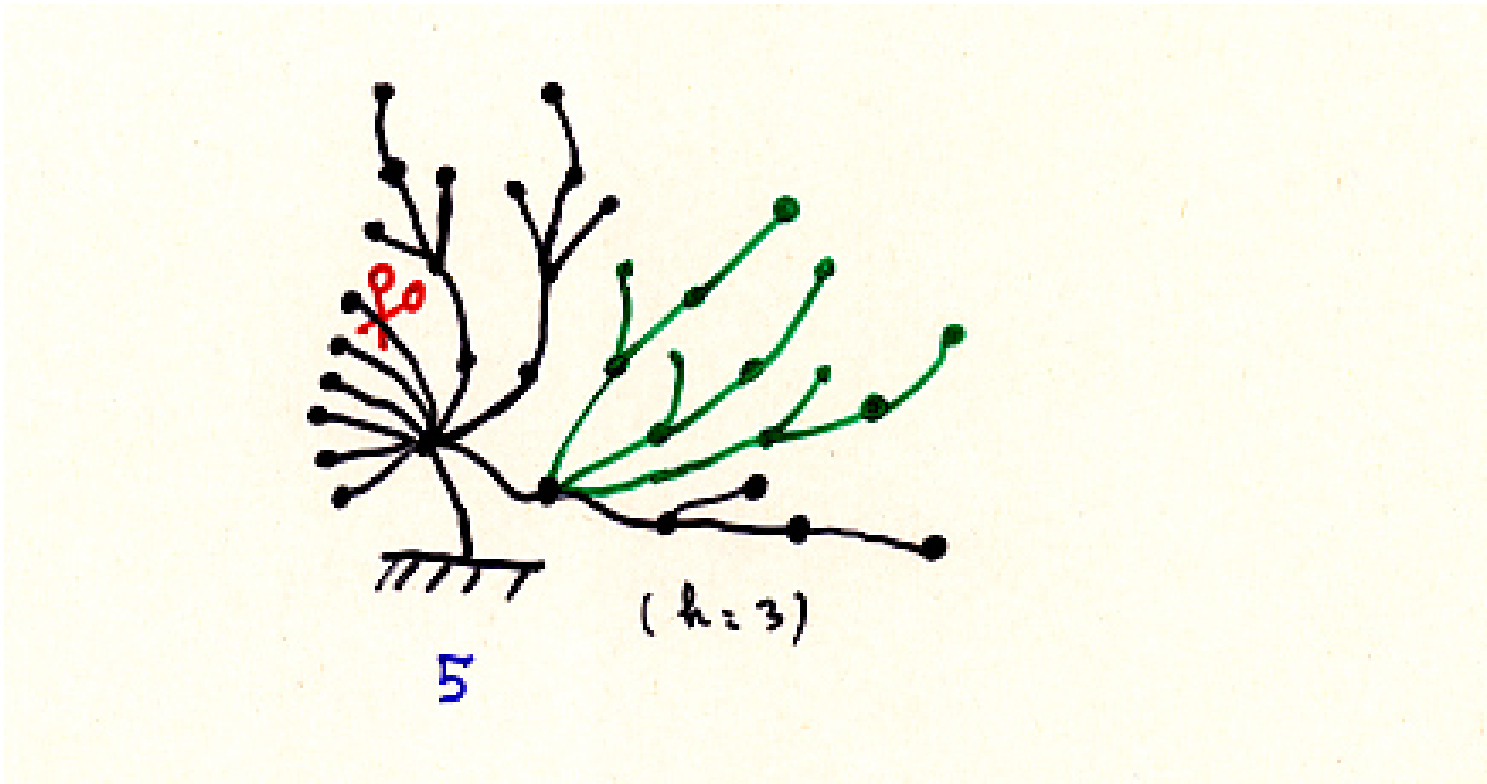
Hercule et l'Hydre



Hercule et l'Hydre (suite)



Hercule et l'Hydre (suite)



Les théorèmes de Kirby et Paris

1. Toute suite de Goodstein finit par atteindre 0
2. Quelques soient les stratégies (récursives) de l'Hydre et d'Hercule, ce dernier finit par vaincre le monstre
3. Ces résultats ne peuvent pas être montrés dans l'arithmétique de Peano

Terminaison de toutes les suites de Goodstein

La récurrence structurelle semble ne pas convenir: Par exemple, si $b = 3$, l'élément suivant b^{b^b} est une somme de 256 termes:

$$
\begin{aligned}
& b^{b^3\times 3+b^2\times 3+b\times 3+3} \times 3 \quad + \\
& b^{b^3\times 3+b^2\times 3+b\times 3+2} \times 3 \quad + \\
& b^{b^3\times 3+b^2\times 3+b\times 3+1} \times 3 \quad + \\
& b^{b^3\times 3+\ b^2\times 3+\ b\times 3} \times 3 \quad + \\
& b^{b^3\times 3+b^2\times 3+b\times 2+3} \times 3 \quad + \quad \ldots \\
& \qquad b^2 \times 3 + b \times 3 + 3
\end{aligned}
$$

(avec $b = 4$).

La preuve de terminaison des suites de Goodstein passe par les étapes suivantes:

- Définir une structure de donnée appropriée pour représenter les items de ces suites
- Utiliser la notion de relation bien fondée.

Dans notre cas, une structure appropriée existe déjà : les ordinaux en forme normale de Cantor (c'est à dire l'ordinal ϵ_0).

Forme normale de Cantor

On peut représenter tous les ordinaux inférieurs à (*i.e.* éléments de) ϵ_0 à l'aide de la forme normale de Cantor:

$$\alpha = \omega^{\alpha_1} \times n_1 + \omega^{\alpha_2} \times n_2 + \ldots \omega^{\alpha_p} \times n_p$$

avec $\alpha > \alpha_1 > \alpha_2 > \cdots > \alpha_p$.
Cette représentation est unique.

Tout élément de la suite de Goodstein peut être représenté par une base b et un ordinal dont les coefficients de la FNC sont inférieurs à b.

$$4^{(4^4+3)} + 4^{4+2} + 4 \times 3$$

$$b^{(b^{b^0}+b^0\times 3)} + b^{b^{b^0}+b^0\times 2} + b^{b^0} \times 3$$

$$\omega^{(\omega^{\omega^0}+\omega^0\times 3)} + \omega^{\omega^{\omega^0}+\omega^0\times 2} + \omega^{\omega^0} \times 3$$

Plan de travail

- Écrire une bibliothèque sur les ordinaux (permettant de représenter au moins ϵ_0), contenant :
  - Une structure de donnée appropriée,
  - Un ordre linéaire $<$
  - Des capacités de calcul: arithmétique, procédures de décision
  - Une preuve de bonne fondation de $<$.
- Prouver que si deux termes consécutifs d'une suite de Goodstein sont de la forme (α, b) et $(\beta, b+1)$, alors $\beta < \alpha$,

Structure de donnée

On représente $\omega^\alpha \times (n+1) + \beta$ par cons α n β.

```
Inductive T1 : Set :=
| cons : T1 → nat → T1 → T1
| zero : T1.

Definition finite n :=
 match n with  0 => zero | S p => cons zero p zero end.

Definition omega := cons (finite 1) 0 zero.

(* ω^ω + ω × 42 *)
Check (cons omega  0 (cons (finite 1) 41 zero)).
```

Un ordre linéaire sur T1

En Prolog approximatif:

```
zero < cons A N B.
cons A N B < cons A' N' B' :- A < A'.
cons A N B < cons A N' B' :- N < N'.
cons A N B < cons A N B' :- B < B'
```

Un ordre linéaire sur T1

```
Inductive lt : T1 → T1 → Prop :=
|  zero_lt : ∀ a n b, zero < cons a n b
|  head_lt :
    ∀ a a' n n' b b', a < a' →
                      cons a n b < cons a' n' b'
|  coeff_lt : ∀ a n n' b b', (n < n')%nat →
                           cons a n b < cons a n' b'
|  tail_lt : ∀ a n b b', b < b' →
                         cons a n b <  cons a n b'

where  "o < o'" := (lt o o') : cantor_scope.
```

Comment prouve-t'on $\alpha < \beta$?

En général par une succession d'applications des constructeurs du type `lt`.

$$\dfrac{\dfrac{\dfrac{\overline{0 < 1}\ \textit{le\_n}}{\omega^0 \times 1 < \omega^0 \times 2}\ \textit{coeff\_lt}}{\omega^1 \times 42 < \omega^2}\ \textit{head\_lt}}{\omega^\omega + \omega \times 42 < \omega^\omega + \omega^2}\ \textit{tail\_lt}$$

Comment prouve-t'on $\alpha < \beta$?

En général par une succession d'applications des constructeurs du type `lt`.

$$\dfrac{\dfrac{\dfrac{\overline{0 < 1}\ \textit{le\_n}}{\omega^0 \times 1 < \omega^0 \times 2}\ \textit{coeff\_lt}}{\omega^1 \times 42 < \omega^2}\ \textit{head\_lt}}{\omega^\omega + \omega \times 42 < \omega^\omega + \omega^2}\ \textit{tail\_lt}$$

```
Lemma L42 : cons omega 0 (cons (finite 1) 41 zero) <
            cons omega 0 (cons (finite 2) 0 zero).
Proof.
 compute; auto with T1 arith.
Qed.
```

Comment prouve-t'on $\alpha < \beta$?

En général par une succession d'applications des constructeurs du type `lt`.

$$\dfrac{\dfrac{\dfrac{\overline{0 < 1}\ \textit{le\_n}}{\omega^0 \times 1 < \omega^0 \times 2}\ \textit{coeff\_lt}}{\omega^1 \times 42 < \omega^2}\ \textit{head\_lt}}{\omega^\omega + \omega \times 42 < \omega^\omega + \omega^2}\ \textit{tail\_lt}$$

```
L42 =
 tail_lt omega 0
     (head_lt 41 0 zero zero
         (coeff_lt zero zero zero (le_n 1)))
```

Comparaison de deux ordinaux

- Motivations:
  - Intervient dans l'arithmétique des ordinaux, fonctions `max`, `+`, `*`, etc.
  - Intervient dans les preuves de terminaison,
  - Construction de tactiques de preuve de $\alpha < \beta$
- Notons que l'écriture ci-dessous n'a pas de sens en *Coq*:

```
if α < β
then ...
else ...
```

Test de comparaison

L'ordre $<$ est total (et décidable).

```
Definition trichotomy_inf :
 ∀ a b,  {a < b} + {a = b} + {b < a}.
...
Defined.
```

Valeurs retournées :

$$\texttt{inleft}\ (b < a)\ (\texttt{left}\ (a = b)\ (\pi : a < b))$$
$$\texttt{inleft}\ (b < a)\ (\texttt{right}\ (a < b)\ (\pi : a = b))$$
$$\texttt{inright}\ (\{a < b\}+\{a = b\})\ (\pi : b < a)$$

Construction interactive de trichotomy_inf

```
Definition trichotomy_inf :
 ∀ a b,  {a < b} + {a = b} + {b < a}.
induction a; destruct b; auto with T1.
```

```
  IHa1 : ∀ b : T1, {a1 < b} + {a1 = b} + {b < a1}

  IHa2 : ∀ b : T1, {a2 < b} + {a2 = b} + {b < a2}
  ============================
   {cons a1 n a2 < cons b1 n0 b2} +
   {cons a1 n a2 = cons b1 n0 b2} +
   {cons b1 n0 b2 < cons a1 n a2}
```

Analyse par cas avec:

- `case (IHa1 b1)`
  cas étudiés : `a1<b1`, `a1=b1` et `b1<a1`,
- Sous l'hypothèse `a1=b1` :
  `case (lt_eq_lt_dec n n0)`
  cas étudiés : `n<n0`, `n=n0` et `n0<n`,
- Sous les hypothèses `a1=b1` et `n=n0` :
  `case (IHa2 b2)`
  cas étudiés : `a2<b2`, `a2=b2` et `b2<a2`.

Dans la plupart des cas, la spécification forte de `trichotomy_inf` permet d'utiliser la tactique `auto with T1`.

Elimination de trichotomy_inf

On peut utiliser `trichotomy_inf` dans des calculs:

```
Definition max a b :=
   if trichotomy_inf a b then b else a.

Goal max ω (ω + 3) = ω + 3.
Proof.
  trivial.
Qed.
```

Elimination de trichotomy_inf

On peut utiliser `trichotomy_inf` dans des calculs:

```
Definition max a b :=
   if trichotomy_inf a b then b else a.

Lemma max_lt_rw : ∀ a b, a < b → max a b = b.
Proof.
 intros a b H;  unfold max;
   case (trichotomy_inf a b); auto.
 intro;case (lt_not_gt H);auto.
Qed.
```

Autre exemple: définition de l'addition (extrait)

- $\mathit{zero} + \alpha = \alpha + \mathit{zero} = \alpha$
- $(\mathtt{cons}\ \alpha\ n\ \beta) + (\mathtt{cons}\ \alpha'\ n'\ \beta') =$

$$\begin{cases} \mathtt{cons}\ \alpha'\ n'\ \beta' & : \quad \alpha < \alpha' \\ \mathtt{cons}\ \alpha\ n\ (\beta + \mathtt{cons}\ \alpha'\ n'\ \beta') & : \quad \alpha' < \alpha \\ \mathtt{cons}\ \alpha\ (n + n' + 1)\ \beta' & : \quad \alpha = \alpha' \end{cases}$$

Spécification faible vs spécification forte

Une autre approche: utiliser une comparaison faiblement spécifiée (ne construit pas de terme de preuve):

```
compare : T1 → T1 → comparison

Inductive comparison : Set :=
    | Eq : comparison
    | Lt : comparison
    | Gt : comparison
```

- gain en temps et espace
- nécessité de prouver la correction de `compare`

```
Fixpoint compare (c c':T1){struct c'}:comparison :=
  match c,c' with
    zero, zero ⇒ Eq
  | zero, cons a' n' b' ⇒ Lt
  | _   , zero ⇒ Gt
  | (cons a n b),(cons a' n' b') ⇒
      (match compare a a' with
      | Lt ⇒ Lt
      | Gt ⇒ Gt
      | Eq ⇒ (match lt_eq_lt_dec n n' with
                | inleft (left _) ⇒ Lt
                | inright _       ⇒ Gt
                |   _             ⇒ compare b b'
              end)
      end)
 end.
```

Compare vs. comparison

Un terme de la forme `compare` t t' se *réduit* soit en `Lt`, soit en `Eq` soit en `Gt`.

```
Eval compute in
    (compare ((finite 2)^omega)
             (omega+omega)).

      = Lt
      : comparison
(* 2^ω  <  ω + ω *)
```

On peut alors définir max de façon plus efficace (sans construction de preuve intermédiaire).

```
Definition max a b := match compare a b with
                      | Lt ⇒ b
                      | _ ⇒ a
                      end.
```

On définit de même les opérations +, *, ^, à l'aide de compare.
Correction de max? Propriétés de ces opérations ?

Il faut établir une relation entre `trichotomy_inf` et `compare`.

```
Theorem compare_reflectR : ∀ a b : T1,
  (match trichotomy_inf a b with
    inleft  (left _) ⇒ Lt
  | inleft  (right _) ⇒ Eq
  | inright _ ⇒ Gt
  end)
  = compare a b.
```

Une tactique pour prouver une inégalité

```
Lemma lt_intro : ∀ a b, compare a b = Lt → a < b.
Proof.
  intros a b; rewrite <- compare_reflectR.
  case (trichotomy_inf a b);auto with T1.
  destruct s; auto.
  discriminate 1.
  discriminate 2.
Qed.

Ltac lt_tac := apply lt_intro;simpl;auto.
```

```
Lemma L4' : ω^ω^ω + ω^ω + ω^(finite 5)
            + ω^(finite 2) + finite 678 <
             ω^ω^ω + ω^ω + ω^(finite 5)
            + ω^(finite 2) + ω.
 lt_tac.
Qed.
Print L4'.

L4' =
lt_intro
  (ω ^ ω ^ ω + ω ^ ω
   + ω ^ finite 5 + ω ^ finite 2 + finite 678)
  (ω ^ ω ^ ω + ω ^ ω
   + ω ^ finite 5 + ω ^ finite 2 + ω)
  (refl_equal Lt)
```

Réflexion

Ce type de schéma est au coeur des *preuves par réflexion*.
Autre exemple: remplacer des preuves de chaînes d'égalités $t_1 = t_2 = \ldots t_n$ par l'application de refl_equal à $\phi(t_1)$, avec ϕ telle que si $t = t'$ alors $\phi(t)$ et $\phi(t')$ sont convertibles.

- Coût ($\times 1$) : construction de ϕ; preuve de la propriété ci-dessus,
- Gain ($\times n$) : partage de termes de preuve (temps et espace)

Pauillac,

dimanche 29 janvier,
9 heures du matin . . .

Résumé de l'épisode précédent

- On peut définir une structure de donnée représentant un bon nombre d'ordinaux.
- Ce sont des arbres définis à partir de deux constructeurs:

  zero: T1

  cons: T1→nat→T1→T1
- On définit sur T1 un ordre total et décidable
- Jusqu'ici, tout allait bien . . .
- On remarque que l'arithmétique sur ces nombres est bizarre: $1 + \omega = \omega = 2^{\omega}$. Ça ne va pas nous faciliter les choses . . .

Bonne fondation d'ϵ_0

Définitions usuelles:

- Une relation $<$ est bien fondée sur A s'il n'existe pas de suite infinie $a_1 > a_2 > \cdots > a_n$,
- Une relation $<$ est bien fondée sur A si toute partie non vide de A admet un élément minimal,

Définition intuitionniste

Accessibilité: Soit R une relation binaire. On définit le prédicat "être accessible" :

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

"x est accessible si tous ses antécédents par R sont accessibles"

- Principe d'induction `Acc_rect`

$$\frac{\forall\, x, (\forall\, y,\ y\ R\ x \Rightarrow \mathrm{Acc}_R(y) \Rightarrow P(y)) \Rightarrow P(x)}{\forall\, x,\ Acc_R(x) \Rightarrow P(x)}$$

- Relation bien fondée: R est bien fondée si tous les éléments de A sont accessibles par R
- Induction bien fondée

$$\frac{\forall\, x, (\forall\, y,\ y\ R\ x \Rightarrow P(y)) \Rightarrow P(x) \quad \mathrm{well\_founded}(R)}{\forall\, x, P(x)}$$

Il nous reste alors à prouver l'accessibilité de tout ordinal $\alpha < \epsilon_0$, c'est à dire de tout terme de `T1` en forme normale.

Termes bien formés

L'ordre $<$ n'est pas bien fondé sur `T1`!

```
Fixpoint f (i:nat) : T1 :=
 match i with
 | 0 ⇒ cons (cons zero 1 zero) 0 zero
 | S p ⇒ cons (cons zero 0 zero) 0 (f p)
 end.
```

On prouve $\forall i,\ f(i+1) < f(i)$.
L'algèbre libre `T1` contient trop d'éléments. On se restreindra alors au sous-ensemble des termes de `T1` *bien formés*

```
Inductive nf : T1 → Prop :=
| zero_nf :  nf zero
| single_nf : ∀ a n, nf a →  nf (cons a n zero)
| cons_nf : ∀ a n a' n' b, a' < a →
                           nf a →
                           nf(cons a' n' b)→
                           nf(cons a n (cons a' n' b)).
Hint Resolve zero_nf single_nf cons_nf : T1.
```

Premier essai

Par induction structurelle sur le type `T1`.

- 0 est accessible
- Soient α et β accessibles, et o=cons α n β en forme normale. Montrons que o est accessible.

Premier essai

Par induction structurelle sur le type `T1`.

- 0 est accessible
- Soient α et β accessibles, et o=cons α n β en forme normale. Montrons que o est accessible.

Raté !

Parmi les antécedents du terme cons α n β, il en est de la forme cons α' n' β', avec $\beta < \beta' < \omega^\alpha$ (et ni $\beta' < \alpha$ ni $\beta' < \beta$).
L'hypothèse de récurrence est donc inexploitable.

La triple induction

Lemme :

$$\forall \alpha,\ \mathrm{Acc}(\alpha) \Rightarrow \mathit{nf}(\alpha) \Rightarrow \underbrace{\forall n\ \beta,\ \mathit{nf}(\beta) \Rightarrow \beta < \omega^{\alpha} \Rightarrow \mathrm{Acc}(\mathtt{cons}\ \alpha\ n\ \beta)}_{Q(\alpha)}$$

Preuve (par récurrence sur l'accessibilité de α): Soit α accessible et en forme normale, tel que

$$\forall \alpha', \mathit{nf}(\alpha') \Rightarrow \alpha' < \alpha \Rightarrow Q(\alpha')$$

(1) Pour tout β en forme normale tel que $\beta < \omega^{\alpha}$, on a $\mathrm{Acc}(\beta)$. (En effet β s'écrit sous la forme $\mathtt{cons}\ \alpha'\ n'\ \beta'$, avec $\beta' < \omega^{\alpha'}$ et $\alpha' < \alpha$.)

La triple induction

Lemme :

$$\forall \alpha,\ \mathrm{Acc}(\alpha) \Rightarrow \mathit{nf}(\alpha) \Rightarrow \underbrace{\forall n\ \beta,\ \mathit{nf}(\beta) \Rightarrow \beta < \omega^{\alpha} \Rightarrow \mathrm{Acc}(\mathtt{cons}\ \alpha\ n\ \beta)}_{Q(\alpha)}$$

Preuve (par récurrence sur l'accessibilité de α): Soit α accessible et en forme normale, tel que

$$\forall \alpha', \mathit{nf}(\alpha') \Rightarrow \alpha' < \alpha \Rightarrow Q(\alpha')$$

(2) On finit la preuve du lemme (par récurrence sur n et sur l'accessibilité de β fraîchement établie.)

La triple induction (fin)

On termine la preuve par récurrence structurelle:

- zero est trivialement accessible,
- Le lemme précédent permet de montrer que si α et β sont accessibles, et si `cons` α n β est en forme normale, alors ce dernier terme est accessible. Notons que l'accessibilité de β est "reprouvée" dans le lemme.

Conséquences On obtient la récurrence et la récursion transfinies.

Réminiscences Cette preuve rappelle par sa structure la preuve de normalisation forte du λ-calcul simplement typé (termes réductibles). Voir aussi les preuves de terminaison des *po.

Toute suite de Goodstein est finie

On associe à tout item de la suite de Goodstein un ordinal (inférieur à ϵ_0) tel que cette suite est strictement décroissante.

$$
\begin{array}{cc}
2 & \omega^3 = \omega^{\omega+1} \\
3 & \omega^\omega \times 2 + \omega^2 \times 2 + \omega \times 2 + 2 \\
4 & \omega^\omega \times 2 + \omega^2 \times 2 + \omega \times 2 + 1 \\
5 & \omega^\omega \times 2 + \omega^2 \times 2 + \omega \times 2 \\
\multicolumn{2}{c}{\dots} \\
N & \omega^\omega \times 2 \\
N+1 & \omega^\omega + \sum_{i=N}^{0} \omega^i \times N
\end{array}
$$

Preuve de la victoire d'Hercule

On associe à toute Hydre H un ordinal $o(H) < \epsilon_0$:

- A toute tête est associé 0
- A toute hydre de la forme $\bullet(H_1, H_2, \ldots, H_n)$, on associe la somme naturelle (associative, commutative et strictement monotone) $\omega^{o(H1)} \oplus \omega^{o(H_2)} \ldots \oplus \omega^{o(H_n)}$.

On prouve que $o(H)$ décroît strictement à chaque étape. Cette preuve utilise des multi-ensembles d'ordinaux.

Les deux propriétés précédentes sont entièrement prouvées en *Coq*, à l'aide d'une bibliothèque sur l'ordinal ϵ_0:

- Forme normale de Cantor
- Preuve de bonne fondation
- Multi-ensembles d'ordinaux
- Lemmes sur l'arithmétique des ordinaux

Exercice: Utiliser la bibliothèque `Color` pour obtenir une preuve directe de la bonne fondation d'ϵ_0, et de la victoire d'Hercule.

Coq et les maths

On souhaite adapter à *Coq* la présentation axiomatique par K. Schütte. de l'ensemble des ordinaux dénombrables (Objectif: comparaison avec les systèmes de notation):

- On considère un ensemble bien ordonné $(\mathbb{O}, <)$, tel que
  - Tout sous-ensemble de $\mathbb{O}$ borné est dénombrable
  - Tout sous-ensemble dénombrable de $\mathbb{O}$ est borné

Comment traduire ces définitions en *Coq*?

Le type Ensemble (module Ensembles)

```
Require Import Ensembles.

Check Ensemble.
Ensemble : Type → Type

Check (Ensemble nat).
Ensemble nat: Type.

Check (Ensemble Prop).
Ensemble Prop : Type.

Check (Ensemble Type).
Ensemble Type^i : Type^j. (i < j)
```

Quelques définitions

```
Definition Included (U:Type) (B C : Ensemble U) :=
 ∀ x : U, In B x → In C x


Definition Same_set (U:Type)(B C:Ensemble U) : Prop :=
        Included B C ∧ Included C B.

Axiom Extensionality_Ensembles :
         ∀ A B:Ensemble, Same_set A B → A = B.

Hint Resolve Extensionality_Ensembles.
```

Quelques définitions (suite)

```
Definition is_least_member (U:Type)
                           (X:Ensemble U)
                           (Le : relation U)
                           (a:U) :=
    In X a ∧
    ∀ x, In X x → Le a x.
```

Structures mathématiques et enregistrements

```
Record WO: Type:= {
 M:Type;
 Lt : relation M;
 D : Ensemble M ;
 Lt_trans : transitive _ Lt;
 Lt_irreflexive : ∀ a:M, ~ (Lt a a);
 well_order : ∀ (MO:Ensemble M)(a:M),
                  In D a → In MO a →
                  ∃ a0:M, is_least_member  D Lt MO a0
}
```

La définition précédente provoque la création de diverses constantes:

- Un constructeur :Build_WO, lequel prend également des preuves en argument
- Des sélecteurs :

```
        M:  WO → Type
       Lt: ∀ w :  WO, relation (M w),
Lt_trans: ∀ w :  WO, transitive (M w) (Lt w)
well_order: ∀ (MO:Ensemble M)(a:M),
            In D a → In MO a →
            ∃ a0:M, is_least_member  D Lt MO a0
      etc.
```

Intuitionnisme ou classique?

On peut souhaiter utiliser des définitions mathématiques usuelles (communément admises, en tout cas plus que la notion d'accessibilité).

> *$(E, <)$ est bien ordonné ssi tout sous-ensemble non vide de E admet un plus petit élément*

Peut-on utiliser cette notion en *Coq* (dans le cadre intuitionniste habituel)?

Eh bien, pas vraiment

- Considérons une preuve π de bonne fondation de l'ordinal $\omega + 1$
- Si cette preuve est constructive, on peut lui associer (par extraction) une fonction f qui à tout sous-ensemble non vide $F \subseteq E$ associe le plus petit élément de F.

- Soit e un entier naturel quelconque, on lui associe l'ensemble (récursif) T_e des ordinaux de $\mathbb{N} \cup \omega$ défini par

  $\alpha \in T_e$ *si*

  - $\alpha = \omega$, *ou*
  - $\alpha \in \mathbb{N}$ *et la machine de code* e *s'arrête après* n *pas sur l'entrée* e.

- L'application de f à T_e nous permet de résoudre élégamment le problème de l'arrêt des machines de Turing.

Merci à Bas Spitters, qui a donné cet argument dans le Coq-Club.

La logique classique en Coq

Le tiers exclu :

```
Axiom classic : ∀ P:Prop, P ∨ ~ P.
```

Conséquences: Certaines inférences deviennent valides :

$$\frac{\sim\sim P}{P}\ NNPP$$

$$\frac{\sim(\forall x, \sim P(x))}{\exists x, P(x)}$$

Logique classique et relations bien fondées

Considérons les trois définitions suivantes:

1. Tout ensemble non vide admet un élément minimal,
2. Il n'existe aucune suite infinie strictement décroissante,
3. tout élément est accessible.

$2 \Rightarrow 3$ si l'on admet l' "axiome du choix fonctionnel":

$$\frac{\forall x, \exists y, x\, R\, y}{\exists f,\ \forall x,\ x\, R\, f(x)}$$

Le reste des implications est prouvable en logique classique.

Exemple de preuve en logique classique

```
Require Import Classical.
Section Clas.
 Variable P : nat → Prop.
 Hypothesis H : ~ (∀ x, ~ ( P x)).

 Lemma E : exists x, P x.
 Proof.
  apply NNPP; firstorder.
 Qed.
End Clas.
```

Comme l'extraction efface les termes de sorte `Prop`, la synthèse de programmes à partir de spécifications reste correcte.

Danger! Axiomes!

```
Require Import Arith.

Record rat_pos : Set :=
  num : nat;
  den : nat;
  den_OK: 0 < den.

Axiom rat_pos_eq : ∀ q q',
   num q * den q' = num q' * den q → q = q'.
```

```
Lemma rat_inj : ∀ q q', q = q' → num q = num q'.
Proof.
 destruct q; destruct q'.
 injection 1; simpl;auto.
Qed.

Lemma L23:  2=3.
Proof.

 Let q1 : rat_pos.
  refine (Build_rat_pos  2 6 _); auto with arith.
 Defined.
```

```
Lemma rat_inj : ∀ q q', q = q' → num q = num q'.
Proof.
 destruct q; destruct q'.
 injection 1; simpl;auto.
Qed.

Lemma L23:  2=3.
Proof.

 ...
 Let q2 : rat_pos.
  refine (Build_rat_pos  3 9 _); auto with arith.
 Defined.
```

```
============================
 2 = 3

 change (num q1 = num q2). (* règle de conversion! *)

 apply rat_inj; apply rat_pos_eq.
 compute;trivial.
Qed.
```

Solution : Utiliser une relation d'équivalence au lieu de l'égalité (Sétoïdes).

L'opérateur de Hilbert

```
Require Import Classical.

Parameter OT : Type ...


Lemma zero_exists : ∃ z, ordinal z ∧
                                ∀ a, ordinal a → z ≤ a.
...
Qed.
```

On souhaite utiliser ce résultat pour définir la constante zero.

```
Definition zero : OT.
 case zero_exists; intros z Hz; exact z.

Toplevel input, characters 5930-5946
> case zero_exists; intros z Hz; exact z.
> ^^^^^^^^^^^^^^^^
Error:
Non Dependent case analysis on sort: Type
is not allowed for inductive definition: ex
```

En effet, admettre d'extraire un témoin d'une preuve d'existence serait en contradiction avec l'irrelevance de preuves (optionnelle en Coq, et de toutes façons dérivable des axiomes de la logique classique).

Idée ?

L'opérateur ϵ de Hilbert :

" Un entier p tel que $p^2 = n$"

```
Parameter epsilon : ∀ (A:Type), (A→Prop) → A.
```

```
Axiom epsilonax  : ∀ (A:Type)  (P : A → Prop),
                     (exists a, P a) →
                    P (epsilon  P).
```

```
Definition zero := epsilon zero_exists.
```

Argh !

```
Lemma L13 : 1 = 3.
 case (epsilon  (fun x:{n:nat | False} ⇒ True)).

1 subgoal

  ============================
   nat → False → 1 = 3

 contradiction.
Qed.
```

Solution?

```
Definition inhabited (A:Type) := ∃ a:A, True.

Parameter epsilon : ∀ (A:Type),
         inhabited A → (A→Prop) → A.

Definition iota (A:Type)(inh : inhabited A)(P: A→Prop)
   :=   epsilon inh
              (fun x ⇒ P x ∧ ∀ y, P y → x = y).

Definition epsilonax  (A:Type) (inh:inhabited A):=
                    ∀ (P : A → Prop),
                      (∃ a, P a) →
                      P (epsilon inh P).
```

Danger! Axiomes! (Rappel)

L'introduction de l'opérateur de choix ϵ peut poser quelques problèmes (travaux de Chicli-Pottier-Simpson, Herman Geuvers).

En premier lieu, on peut dériver du tiers exclu (logique classique) la décidabilité de tout prédicat.

```
Require Import Iota.
Require Import Classical.

Section R_declared.
 Variable A : Set.
 Variable P : A → Prop.
 (* On définit une relation binaire entre les types
 A et {P a}+{~ P a} *)

 Inductive  R (a:A) :{P a}+{~ P a} → Prop :=
 | caseA : ∀ H : P a , R  a (left (~ P a) H)
 |  caseB : ∀ H : ~ P a , R  a (right ( P a) H).

 Hint Constructors R.
```

```
 (* Pour utiliser iota, on s'assure que le type
({P a}+{~ P a}) est habité *)
Lemma R_inh : ∀ a, inhabited ({P a}+{~ P a}).
Proof.
 intro a; case (classic (P a)).
 intro H;exists (left (~ P a) H); auto.
 intro H;exists (right ( P a) H); auto.
Qed.
```

```
(* Le mal est fait ... *)
Definition toto (a : A) : {P a} + {~ P a}.
intro.
 exact (epsilon (R_inh a) (R a)).
Defined.

End R_declared.
toto :
  ∀ (A : Set) (P : A → Prop) (a : A),
    {P a} + {~ P a}
```

Tout prédicat deviendrait décidable si l'on utilisait l'extraction

Remarquons que c'est la déclaration d'ϵ qui pose ce problème (l'axiome d'élimination n'est pas utilisé dans cette dérivation).

Solutions

- inspecter les termes de preuves, à la recherche des axiomes utilisés,
- Utiliser à bon escient le système de modules: un module contenant une extraction ne doit pas importer `Classical + Iota`.
- Lire les warnings:

```
Extraction toto.
Warning: You must realize axiom epsilon in the ex-
tracted code.
(** val toto : 'a1 → sumbool **)
let toto a =
  epsilon __
```

[Im]prédicativité de Set?

- Prédicativité de Set par défaut

```
Check (∀ (A:Set), list A → list A).
∀ A : Set, list A → list A
     : Type
```

- Imprédicativité de Set option -impredicative-set

```
Check (∀ (A:Set), list A → list A).
∀ A : Set, list A → list A
     : Set
```

Prop est imprédicative

```
∀ P Q R:Prop, (P→Q)→(Q→R)→P→R  : Prop

Definition my_False : Prop := ∀ P:Prop, P.

Definition impred_eq (A:Type)(a b:A) : Prop :=
    ∀ P : A→ Prop, P a → P b.

Theorem impred_eq_sym : ∀ (A:Type)(a b:A),
            impred_eq _ a b → impred_eq _ b a.
Proof.
  intros A a b H P; red in H.
  apply H; trivial.
Qed.
```

Qu'en est-il de Type ?

```
Check (
∀ (M : Type(i)) (Lt : relation M) (D : Ensemble M),
       (∀ x y : M, Lt x y → In D x) →
       (∀ x y : M, Lt x y → In D y) →
        transitive M Lt →
       (∀ a : M, ~ Lt a a) →
       (∀ (M0 : Ensemble M) (a : M),
        In D a → In M0 a →
                 ∃ a0 : M, is_least_member D Lt M0 a0) →
        WO)

Type(j).
```

Paradoxe de Hurkens, adapté par H. Geuvers

> *Dans la version de Coq avec* `Set` *imprédicatif, l'axiome*
>
> $\forall$ `P:Prop, {P}+{~P}`
>
> *est incohérent.*
>
> *H. Geuvers : Inconsistency of classical logic in type theory*

On en conclut que la conjonction:

- tiers exclu
- existence d'ϵ
- option -impredicative-set

mène droit à l'incohérence.

Extrait de la *FAQ* de *Coq*

The axiom of description together with classical logic (e.g. excluded-middle) are inconsistent in the variant of the Calculus of Inductive Constructions where Set is impredicative.
As a consequence, the functional form of the axiom of choice and excluded-middle, or any form of the axiom of choice together with predicate extensionality are inconsistent in the Set-impredicative version of the Calculus of Inductive Constructions.

Extrait de la *FAQ* de *Coq*

The main purpose of the Set-predicative restriction of the Calculus of Inductive Constructions is precisely to accommodate these axioms which are quite standard in mathematical usage.
The Set-predicative system is commonly considered consistent by interpreting it in a standard set-theoretic boolean model, even with classical logic, axiom of choice and predicate extensionality added.

Conclusion

- On a eu une vision très partielle de *Coq*.
- On espère avoir pu montrer que c'est intéressant.

Pour vous y mettre:

- Tutoriaux, documentation et Faq en ligne: `coq.inria.fr`,
- Le Coq'Art, *et son site d'exercices corrigés*,
- L'abonnement (gratuit) au Coq-Club: coq-club@pauillac.inria.fr,
- Wiki "Cocorico",
- Lecture des contributions.