Définitions inductives

Hugo Herbelin

Contenu

- Exemples de types inductifs
- Conditions de positivité
- Schémas d'éliminations (dépendance, élimination forte)
- Tactiques spécifiques (EqDecide/Discriminate/Injection/Inversion)
- Récursion structurelle/Récursion générale
- Utilisation de Auto/EAuto (ajout de Hint dans la base)

Travaux Pratiques

- 1ère partie: exos spécifiques (arithmétique...)
- 2ème partie: formalisation d'un langage impératif parallèle

Retour sur les naturels

```
Coq < Inductive nat : Set := O : nat
Coq <                      | S : nat -> nat.
```

À comparer avec la syntaxe Caml :

```
type nat = O | S of nat ;;
```

Effets d'une déclaration inductive

Ajoute le type et les constructeurs dans l'environnement :

```
Coq < Check nat.
nat
      : Set
Coq < Check O.
O
      : nat
Coq < Check S.
S
      : nat->nat
Coq < Check nat_ind.
nat_ind
      : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```


Effets d'une déclaration inductive

Autorise le filtrage sur `O` et `S` :

```
Coq < Definition is_zero [n] := Cases n of O => true | (S _) => false end.
```

Et engendre les principes d'induction associés

```
Coq < Check nat_ind.
nat_ind
      : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
Coq < Check nat_rec.
nat_rec
      : (P:(nat->Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
Coq < Check nat_rect.
nat_rect
      : (P:(nat->Type))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```


Des fonctions définies par récursion

```
Coq < Fixpoint plus [n:nat] : nat -> nat :=
Coq <   Cases n of
Coq <      O => [m:nat]m
Coq <    | (S p) => [m:nat](S (plus p m))
Coq <  end.
plus is recursively defined
```

Terminaison: récurrence structurelle.

Des fonctions définies par récursion

Mais :

```
Coq < Fixpoint boucle [n:nat] : nat := (boucle n).
Error:
Recursive call applied to an illegal term
The recursive definition boucle := [n:nat](boucle n) is not well-formed
```

Le système refuse les fonctions qui ne terminent pas.

Pourquoi se restreindre aux fonctions prouvablement terminantes

1. On pourrait montrer l'absurde...

```
Coq < Fixpoint Boucle [n:nat] : False := (Boucle n).

Coq < Theorem inconsistency : False.

Coq < Exact (F O).

Coq < Qed.
```

2. On ne pourrait plus décider la typabilité car celle-ci est considérée modulo réduction.

Terminaison des fonctions définies par récurrence

- Le problème est indécidable.
- Le critère de Coq : les appels récursifs doivent se faire sur des arguments *structurellement* plus petits

Exemple :

```
Coq < Fixpoint d2 [n:nat] : nat :=
Coq <   Cases n of O         => O
Coq <            | (S O)     => O
Coq <            | (S (S p)) => (S (d2 p))
Coq <   end.
d2 is recursively defined
```

Mais alors, comment prouver la terminaison lorsque la décroissance n'est pas structurelle ? On en reparlera.

Autre type de données : les listes

On retrouve les mêmes principes.

```
Coq < Require PolyList.

Coq < Print list.
Inductive list [A : Set]  : Set :=
      nil : (list A) | cons : A->(list A)->(list A)
For cons: Position [1] is implicit

Coq < Check list_ind.
list_ind
     : (A:Set; P:((list A)->Prop))
        (P (nil A))
        ->((a:A; l:(list A))(P l)->(P (cons a l)))
        ->(l:(list A))(P l)
```


Condition de positivité

Pour que les types des constructeurs correspondent bien à une définition mathématique, on exige que les arguments récursifs soient *positifs*.

Concrètement, la définition suivante est refusée:

```
Coq < Inductive foo : Set :=
Coq <  C : (foo->foo)->foo.
Error: Non strictly positive occurrence of foo in (foo->foo)->foo
```

- mathématiquement, il n'y a pas de *plus petit ensemble* vérifiant cette condition de cloture,
- par ailleurs on perd la terminaison:

```
Definition f : foo -> foo := [x]Cases x of (C g) => (g (C g)) end.
```

`(f (C f))` boucle

Généralité des définitions inductives

En revanche, la définition suivante est correcte:

```
Inductive ord : Set :=
  Oo : ord
| So : ord -> ord
| Lim : (nat->ord)->ord.
```

On sort de la notion de *type de donnée*

Quelque soit `n:nat`, `(f n)` est structurellement plus petit que `(Lim f)`.

Le produit et la paire

```
Coq < Inductive prod [A:Set; B:Set] : Set :=
Coq <                  pair : A->B->(prod A B).
```

pretty-printé `A*B` et `(a,b)`.

Non récursif, mais élimination tout de même:

```
Coq < Check prod_ind.
prod_ind
     : (A,B:Set; P:((prod A B)->Prop))
        ((a:A; b:B)(P (pair A B a b)))->(p:(prod A B))(P p)
```


Définitions inductives des connecteurs logiques

```
Coq < Inductive and [A:Prop; B:Prop] : Prop :=
Coq <                      conj : A->B->(and A B).
```

```
Coq < Check and_ind.
and_ind
     : (A,B,P:Prop)(A->B->P)->(and A B)->P
```

Très semblable au produit...

On voit aussi qu'une preuve canonique de `A/\B` est de la forme `(conj A B a b)`

```
Coq < Print and_ind.
and_ind =
[A,B,P:Prop](and_rect A B P)
     : (A,B,P:Prop)(A->B->P)->(and A B)->P
```


Définitions inductives des connecteurs logiques

La disjonction...

```
Coq < Print or.
Inductive or [A : Prop; B : Prop]  : Prop :=
      or_introl : A->A\/B | or_intror : B->A\/B
```

```
Coq < Check or_ind.
or_ind
     : (A,B,P:Prop)(A->P)->(B->P)->A\/B->P
```

le quantificateur $\exists$

```
Coq < Inductive ex [A:Set; P:A->Prop] : Prop :=
Coq <       ex_intro : (x:A)(P x)->(ex A P).
```

pretty-printé `(EX x:A | (P x))`.

C'est une *paire dépendante*.

```
Coq < Print ex_ind.
ex_ind =
[A:Set; P:(A->Prop); P0:Prop; f:((x:A)(P x)->P0); e:(ex A P)]
 <P0>Cases e of (ex_intro x x0) => (f x x0) end
     : (A:Set; P:(A->Prop); P0:Prop)((x:A)(P x)->P0)->(ex A P)->P0
```


Qu'est-ce que le faux ?

La proposition fausse $\perp$ est telle que pour toute proposition P on a $\perp \Rightarrow P$.

Deux définitions possibles:

- `Definition False := (P:Prop)P.`
- `Inductive False : Prop := .`

En prenant la proposition inductive sans constructeur, on retrouve:

```
False_ind
     : (P:Prop)False->P
```

Attention ! on distingue

`False : Prop` et `false:bool`

Une proposition vraie

Il est pratique de disposer d'une proposition tout le temps vraie :

```
Coq < Print True.
Inductive True  : Prop :=  I : True
```

Cette fois le principe d'élimination n'est pas très intéressant

```
True_ind
     : (P:Prop)P->True->P
```


L'égalité

La plus petite relation réflexive:

```
Coq < Inductive eq [A:Set; x:A] : A->Prop :=
Coq <                     refl_equal : (eq A x x).
```

pretty-printé `a=b`.

Élimination : on retrouve le *principe de Leibniz*.

```
eq_ind
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)(eq A x y)->(P y)
```

Types mutuellement inductifs

```
Coq < Variable A : Set.
Coq < Inductive tree : Set :=  node : A -> forest -> tree
Coq < with forest : Set := leaf : A -> forest | constree : tree -> forest -> fores
tree, forest are defined
tree_rect is defined
tree_ind is defined
tree_rec is defined
forest_rect is defined
forest_ind is defined
forest_rec is defined
Coq < Check tree_ind.
tree_ind
     : (P:(tree->Prop))((a:A; f:forest)(P (node a f)))->(t:tree)(P t)
```


Schémas d'induction sur mesure

```
Coq < Scheme tree_forest_rec := Induction for tree Sort Set
Coq < with forest_tree_ind := Minimality for forest Sort Prop.
tree_forest_rec, forest_tree_ind are recursively defined
Coq < Check tree_forest_rec.
tree_forest_rec
     : (P:(tree->Set); P0:Prop)
        ((a:A; f:forest)P0->(P (node a f)))
        ->(A->P0)
        ->((t:tree)(P t)->forest->P0->P0)
        ->(t:tree)(P t)
```


Élimination et calcul

Les réductions simplifient les preuves.

```
Coq < Lemma pr
Coq <    : (a:nat)(EX b : nat | a=(plus b b)\/a=(S (plus b b))).
```

une preuve canonique pour `a=3` est de la forme

```
(ex_intro nat [b](...) (S O) (or_intror ... (refl_equal nat (3)))
```


Que se cache derrière `nat_ind` ?

```
Coq < Eval Simpl in (nat_ind P p0 pS O).
     = p0
     : (P O)
Coq < Eval Simpl in [n](nat_ind P p0 pS (S n)).
     = [n:nat](pS n (nat_ind P p0 pS n))
     : (n:nat)(P (S n))
```

Ceci impose la définition de `nat_ind`.

```
Fixpoint nat_ind [P:(nat->Prop); p0; pS; n:nat ] : (P n)
  := Cases n of O => p0
              | (S x) => (pS n (nat_ind P p0 pS x))
              end.
```

Filtrage (dépendant) et récursion permettent de tout faire !

$0 \neq 1$?

C'est le “dernier” axiome de l'arithmétique.

$0 = 1$ signifie que 0 et 1 vérifient les mêmes propriétés

Prouver $0 \neq 1$ c'est exhiber un prédicat P tel que

- $P(0)$ est prouvable
- $\neg P(1)$ est prouvable

Comment faire ?

On veut pouvoir définir la proposition $P(x)$ par cas sur x

Définition de types/propositions par cas

```
Coq < Definition P : nat -> Prop :=
Coq <   [n] Cases n of
Coq <    O => True
Coq < | (S x) => False
Coq < end.
```

- Prouver `(P (O))` est trivial (c'est `True`)
- `(P (1))` c'est `False`
- Si `(O)=(1)` alors on peut passer de l'un à l'autre.

Cette construction est automatisée dans la tactique `Discriminate`

Discriminate

```
Coq < Lemma nil_cons : (A:Set)(a:A)(m:(list A))~(nil A)=(cons a m).
1 subgoal

  ============================
   (A:Set; a:A; m:(list A))~(nil A)=(cons a m)

Coq < Intros; Discriminate.
Subtree proved!
```


On a aussi l'injectivité des constructeurs

```
Coq < Lemma ex1 : (A:Set)(a,a':A)(m,m':(list A))(cons a m)=(cons a' m')->True.

Coq < Intros A a a' m m' H; Injection H.
1 subgoal

  A : Set
  a : A
  a' : A
  m : (list A)
  m' : (list A)
  H : (cons a m)=(cons a' m')
  ============================
   m=m'->a=a'->True
```

A propos des définitions de types par récurrence

La possibilité de définir un type par cas, comme pour `P`, s'appelle *élimination forte*.

Le type `Prop` des propositions et le type `Set` des types calculatoires étant eux-mêmes typés, et leur type est `Type`, on peut exprimer le schéma de récurrence correspondant :

```
Coq < Check nat_rect.
nat_rect
     : (P:(nat->Type))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```


Prédicats inductifs / prédicats récursifs

On a souvent le choix entre les deux versions d'une définition.

On veut dire qu'un entier est plus petit que les éléments d'une liste:

```
Coq < Inductive minore [n:nat] : (list nat) -> Prop :=
Coq <   minore_nil : (minore n (nil ?))
Coq < | minore_cons : (a:nat)(l:(list ?))(le n a)->
Coq <                   (minore n l)->(minore n (cons a l)).

Coq <
Coq < Fixpoint minore_r [n:nat; l:(list nat)] : Prop :=
Coq <  Cases l of
Coq <    nil => True
Coq < | (cons a r) => (le n a)/\(minore_r n r)
Coq < end.
```


Avantage de la définition inductive: on n'a pas besoin de prouver les clauses.

Avantage de la définition récursive: on prouve facilement que

- `(minore_r n (cons a l))->(le n a)`
- `(minore_r n (cons a l))->(minore_r n l)`

La tactique `Inversion` permet de faire de même pour les définitions inductives (en passant en fait par une définition récursive)

```
1 subgoal

  pl : (minore a (cons x l))
  ============================
   P
```



```
Coq < Inversion pl.
1 subgoal

  pl : (minore a (cons x l))
  a0 : nat
  l0 : (list nat)
  H : a0=x
  H0 : l0=l
  H1 : (le a x)
  H2 : (minore a l)
  ============================
   P
```

ou bien :

```
Coq < Inversion_clear pl.
1 subgoal

  H : (le a x)
  H0 : (minore a l)
  ============================
   P
```


Égalité sur les types de données

Un type de donnée est un type qui n'encapsule aucune fonction.

On peut toujours décider de l'égalité sur un type de donnée.

```
comp : A->A->bool
comp_corr : (a,b:A)(comp a b)=true<->a=b
```

un peu lourd...

Meilleure formulation:

```
eqA_dec : (a,b:A){a=b}+{~a=b}
```

Preuve "à la main" lourde dans le cas général

L'occasion de présenter l'induction double

```
Coq < Lemma ex2 : (m,n:nat){m=n}+{~m=n}.
1 subgoal

  ============================
   (m,n:nat){m=n}+{~m=n}

Coq < Double Induction 1 2; Auto; Intros; Elim (H0 n); Auto.
Subtree proved!
```


Automatisation Decide Equality et Compare

```
Coq < Lemma ex3 : (b1,b2:bool){b1=b2}+{~b1=b2}.
exiv < 1 subgoal

  ============================
   (b1,b2:bool){b1=b2}+{~b1=b2}

Coq < Decide Equality.
Subtree proved!
```

Comparaison “à la volée”

```
1 subgoal

  n : nat
  m : nat
  ============================
   P

Coq < Compare n m.
2 subgoals

  n : nat
  m : nat
  ============================
   n=m->P
subgoal 2 is:
 ~n=m->P
```


Retour sur les définitions par point-fixe

Comment gérer un ordre de décroissance arbitraire ?

Exemple (une variante de la fonction 91) :

$$f_{91}(x) = x + 11 \qquad \text{if } x < 100$$

$$f_{91}(x) = x - 10 \qquad \text{if } x > 100$$

$$f_{91}(100) = 0 \qquad \text{if } x > 100$$

Cette fonction termine mais l’ordre de bonne fondaison n’est pas structurel.

Solution : traduire l’ordre en l’ordre structurel dans un type inductif.

Récurrence structurelle et récurrence générale

L’approche standard proposée dans Coq est de montrer l’accessibilité de tout argument de la fonction selon l’ordre de bonne fondaison.

```
Inductive Acc [A : Set; R : A->A->Prop]  : A->Prop :=
      Acc_intro : (x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x)

well_founded =
[A:Set; R:(A->A->Prop)](a:A)(Acc A R a)
     : (A:Set)(A->A->Prop)->Prop
```


Définir la fonction 91

```
Coq < Lemma l1 : (n:nat)(LE91 (plus n (11)) n).

Coq < Lemma l2 : (n:nat)(LE91 (minus n (10)) n).

Coq < Fixpoint f91 [n:nat;H:(Acc nat LE91 n)] : nat :=
Coq <    Cases H of (Acc_intro _ g) =>
Coq <      if (eq_dec n (100)) then [H](0)
Coq <      else if (le_dec n (100))
Coq <        then [H](f91 (plus n (11)) (g (plus n (11)) (l1 n)))
Coq <        else [H](f91 (minus n (10)) (g (minus n (10)) (l2 n)))
Coq <    end.

Coq < Theorem acc_le : (n:nat)(Acc nat LE91 n).

Coq < Eval Compute (f91 (30) (acc_le (30)).
```

Paramétrisation de Auto

`Auto` : résolution à base de filtrage.

```
lt_le_S
    : (n,p:nat)(lt n p)->(le (S n) p)
le_S_n
    : (n,m:nat)(le (S n) (S m))->(le n m)
```



```
Coq < Lemma ex5 : (m,n:nat)(lt n (S m))->(le n m).

Coq < Auto.
1 subgoal

  ============================
   (m,n:nat)(lt n (S m))->(le n m)

Coq < Hints Resolve lt_le_S.

Coq < Hints Immediate le_S_n.

Coq < Auto.
Subtree proved!
```


Résolution à base d'unification : `EAuto`

```
Coq < Lemma ex6 : (m,n,p:nat)(le n (S m))->(le m p)->(le n (S p)).

Coq < Hints Resolve le_trans.
Warning: the hint: EApply le_trans will only be used by EAuto

Coq < Hints Resolve le_n_S.

Coq < Info EAuto.
 == Intro m; Intro n; Intro p; Intro H; Intro H0;
       EApply le_trans.
    EExact H.

    Apply le_n_S; EExact H0.

Subtree proved!
```


Aspects avancés du langage de spécification

Hugo herbelin & Christine Paulin

Objectifs

- Facilités de notation :
  - Arguments implicites
  - Type Record
  - Coercions
- Contrôle de la réduction des constantes
- Axiomes (logique classique, extensionnalité, égalité dépendante)

Arguments implicites

```
Coq < Type refl_equal.
refl_equal
     : (A:Set; x:A)x=x

Coq < Type (refl_equal O).
Toplevel input, characters 35-36
> Type (refl_equal O).
>                  ^
Error: The term O has type nat while it is expected to have type Set

Coq < Type (refl_equal ? O).
(refl_equal nat O)
     : O=O
```


Mode Implicit Arguments

```
Coq < Set Implicit Arguments.

Coq < Inductive eq2 [A:Set;x:A] : A -> Prop := refl_eq2 : (eq2 x x).

Coq < Type refl_eq2.
refl_eq2
     : (A:Set; x:A)(eq2 x x)

Coq < Print refl_eq2.
Inductive eq2 [A : Set; x : A]  : A->Prop :=
      refl_eq2 : (eq2 x x)
For eq2: Position [1] is implicit
For refl_eq2: Position [1] is implicit

Coq < Type (refl_eq2 true).
(refl_eq2 true)
     : (eq2 true true)
```


Arguments implicites

- Le mode `Implicit Arguments` : plus élégant, automatique
- L'utilisation de "?" : syntaxe plus chargée, contrôle par l'utilisateur

Règle de génération :

Soit $c : (x_1 : A_1) \ldots (x_n : A_n)B$
tout argument i tel que x_i apparaît libre dans A_j est déclaré implicite.

Recalculé au moment de la fermeture de section.

Arguments implicites-limites

```
Coq < Set Implicit Arguments.

Coq < Parameter exist : (P:nat->Prop)(x:nat)(P x)->(EX x:nat | (P x)).
exist is assumed

Coq < Print exist.
*** [ exist :
(P:(nat->Prop); x:nat)(P x)->(EX x0:nat | (P x0)) ]
Positions [1; 2] are implicit

Coq < Type (exist I).
Toplevel input, characters 12-13
> Type (exist I).
>             ^
Error: The term I has type True while it is expected to have type
 (?5 ?6)
```


Le mode `Implicit Arguments`-contrôle

```
Coq < Type (refl_eq2 bool true).
Toplevel input, characters 6-24
> Type (refl_eq2 bool true).
>       ^^^^^^^^^^^^^^^^^^^
Error:
Illegal application (Non-functional construction):
The expression (refl_eq2 bool) of type (eq2 bool bool)
cannot be applied to the term
 true : bool
```

Succès

```
Coq < Type (!refl_eq2 bool true).

Coq < Type (!refl_eq2 ? true).

Coq < Type (refl_eq2 1!bool true).
```


Forcer des arguments implicites

```
Coq < Implicits refl_eq2 [].

Coq < Print refl_eq2.
Inductive eq2 [A : Set; x : A]  : A->Prop :=
      refl_eq2 : (eq2 x x)
For eq2: Position [1] is implicit

Coq < Type (refl_eq2 bool true).
(refl_eq2 bool true)
     : (eq2 true true)

Coq < Implicits refl_equal [1].

Coq < Type (refl_equal true).
(refl_equal true)
     : true=true
```


Utilisation de ?

Un ? à la place d'un terme calculé automatiquement par le typage.

```
Coq < Definition p : nat->nat := [x:?]x.
p is defined

Coq < Definition p1 : nat->nat := [x]x.
p1 is defined

Coq < Print p1.
p1 = [x:nat]x
     : nat->nat

Coq < Definition p2 := [x]x.
Error: There is an unknown subterm I cannot solve

Coq < Definition p3 := [x](plus x O).
p3 is defined
```

Le type enregistrement ("Record")

```
Coq < Require Reals.

Coq < Record complex : Type := {real:R ; imag:R}.
complex is defined
complex_rect is defined
complex_ind is defined
complex_rec is defined
```


Le type enregistrement ("Record")

- Juste du sucre syntaxique
- Pas de notation `{real=a ; imag=b}`
- Pas de notation `c.real`

Le type enregistrement ("Record")

De fait :

```
Coq < Print complex.
Inductive complex  : Type :=  Build_complex : R->R->complex

Coq < Print real.
real =
[c:complex]Cases c of (Build_complex real _) => real end
     : complex->R
```


Exemple : une structure de pile

```
Coq < Parameter A : Set.

Coq < Record abs_stack : Type := {
Coq <   stack : Set;
Coq <   create : stack;
Coq <   empty : stack -> bool;
Coq <   push  : A -> stack -> stack;
Coq <   pop   : stack -> (Exc (A*stack));
Coq <   pop_empty : (s:stack)(empty s)=true->(pop s)=(error (A*stack));
Coq <   pop_push : (s:stack)(a:A)(pop (push a s))=(Value (a,s))
Coq < }.
```

Exemple : une structure abstraite de pile

Problème : `(empty s)` a le type `bool`
on ne peut écrire :
`(empty s)->(pop s)=(error (A*stack))`

Solution 1 : Écrire `(empty s)=true`, c'est une proposition

Solution 2 : Utiliser une coercion implicite de `bool` vers `Prop`

Les coercions implicites

```
Coq < Coercion is_true := [b:bool]b=true.
is_true is defined
is_true is now a coercion

Coq < Type false -> true.
false->true
     : Prop
```

Attention : seule la surcharge de parenthèses indique la présence d'une coercion implicite.

```
Coq < Set Printing Coercion is_true.

Coq < Type false -> true.
(is_true false)->(is_true true)
     : Prop
```


Exemple : une pile à base de listes

```
Coq < Require PolyList.

Coq < Local SLError := (error (A * (list A))).

Coq < Definition create_list := (nil A).

Coq < Definition empty_list := [s:(list A)]
Coq <   Cases s of nil => true | _ => false end.

Coq < Definition push_list := (!cons A).

Coq < Definition pop_list := [s:(list A)]
Coq <   Cases s of nil => SLError
Coq <     | (cons a s) => (Value (a, s)) end.
```


Exemple : une pile à base de listes

```
Coq < Lemma pop_list_empty :
Coq <   (s:(list A))(empty_list s)->(pop_list s)=SLError.

Coq < Induction s; Simpl; Intros; Trivial.

Coq < Discriminate H0.

Coq < Save.

Coq < Lemma pop_list_push :
Coq <   (s:(list A))(a:A)(pop_list (push_list a s))=(Value (a,s)).

Coq < Trivial.

Coq < Qed.
```

Exemple

Instantiation de la structure abstraite par la structure concrète
Quelles sont les positions implicites de `Build_abs_stack` ?

```
Coq < Print Build_abs_stack.
Inductive abs_stack  : Type :=
      Build_abs_stack :
       (stack:Set) stack
    -> (emp:(stack->bool);pus:(A->stack->stack);pop:(stack->(Exc A*stack))
       ((s:stack)(emp s)=true->(pop s)=Error)
    -> ((s:stack; a:A)(pop (push a s))=(Value (a,s)))
    -> abs_stack
For Build_abs_stack: Positions [1;3;4;5] are implicit
```


Il suffit donc de fournir les 2ème , 6ème et 7ème arguments : `create`, `pop_empty`, `pop_push`.

```
Coq < Definition stack_list : abs_stack :=
Coq <  (Build_abs_stack create_list pop_list_empty pop_list_push).
stack_list is defined

Coq < Eval Compute in  (stack stack_list).
     = (list A)
     : Set
```


Exemple (suite)

L'alternative explicitant tous les arguments :

```
Coq < Definition stack_list2 := (!Build_abs_stack
Coq <   (list A)
Coq <   create_list
Coq <   empty_list
Coq <   push_list
Coq <   pop_list
Coq <   pop_list_empty
Coq <   pop_list_push
Coq < ).
stack_list2 is defined
```


Exemple : un théorème sur les piles

On aimerait pouvoir identifier la pile abstraite avec son support.

```
Coq < Variable AS : abs_stack.

Coq < Theorem th1 : (s:AS)(a:A)~(empty (push a s)).
```

$\hookrightarrow$ Introduire une coercion de `abs_stack` vers `(stack abs_stack)`.

Coercions implicites dans les “Record”

```
Coq < Record abs_stack2 : Type := {
Coq <   stack2 :> Set;
Coq <   create2 : stack2;
Coq <   empty2 : stack2 -> bool;
Coq <   push2 :  A -> stack2 -> stack2
Coq < }.
abs_stack2 is defined
abs_stack2_rect is defined
abs_stack2_ind is defined
abs_stack2_rec is defined
```


Exemple : un théorème sur les piles

```
Coq < Parameter AS2 : abs_stack2.
AS2 is assumed
Coq < Theorem th2 : (s:AS2)(a:A)~(empty2 (push2 a s)).
1 subgoal

  ============================
   (s:AS2; a:A)~(is_true (empty2 (push2 a s)))
```


Les coercions implicites

- Les différentes classes d’où et vers où une coercion agit
  - Sortes
  - Types inductifs et constantes
  - Types produits
- Les coercions se composent
- En cas de chemins multiples, le dernier défini est utilisé

Blocage de la réduction des constantes

- `Transparent` ou `Opaque`
- Les théorèmes sont `Opaque` par défaut
- Les définitions sont `Transparent` par défaut

Blocage de la réduction des constantes

```
Coq < Require Arith.

Coq < Eval Compute in (plus (10) (15)).
     = (25)
     : nat

Coq < Opaque plus.

Coq < Eval Compute in (plus (10) (15)).
     = (plus (10) (15))
     : nat
```


L'ajout d'axiomes

- Pas de contenu calculatoire a priori
- Susceptible d'introduire des paradoxes

La logique classique

```
Coq < Require Classical_Prop.

Coq < Print classic.
*** [ classic : (P:Prop)P\/~P ]
```

- Jamais de contenu calculatoire dans Prop
- Entraine l'égalité de toutes les preuves d'une même proposition ("proof-irrelevance")
- Cohérent
- Dans `Set`, `{P}+{~P}` permettrait de montrer `False`.

L'extensionalité

```
Coq < Axiom extensionality :
Coq <    (A,B:Set)(f,g:A->B) ((x:A)(f x)=(g x)) -> f=g.
extensionality is assumed
```

Nécessaire pour des données représentées par des fonctions : environnements, tableaux...

L'égalité dépendante

```
Coq < Parameter listn : nat -> Set.

Coq < Parameter n,m:nat.

Coq < Parameter l1:(listn n).

Coq < Parameter l2:(listn m).

Coq < Axiom diff : (n=m)->(l1=l2)->False.
Toplevel input, characters 24-26
> Axiom diff : (n=m)->(l1=l2)->False.
>                        ^^
Error: In environment
e : n=m
The term l2 has type (listn m) while it is expected to have type
 (listn n)
```


L'égalité dépendante

Une égalité permettant de comparer des termes de type différent :

```
Coq < Require Eqdep.

Coq < Print eq_dep.
Inductive eq_dep [U : Set; P : U->Set; p : U; x : (P p)]
     : (q:U)(P q)->Prop :=  eq_dep_intro : (eq_dep U P p x p x)

Coq < Axiom diff : (n=m)->(eq_dep ? listn ? l1 ? l2)->False.
diff is assumed
```



```
Coq < Theorem dep_eq : (l3:(listn n))(eq_dep ? listn ? l1 ? l3)->l1=l3.
1 subgoal

  ============================
   (l3:(listn n))(eq_dep nat listn n l1 n l3)->l1=l3

Coq < Print eq_rec_eq.
*** [ eq_rec_eq :
(U:Set; p:U; Q:(U->Set); x:(Q p); h:(p=p))x=(eq_rec U p Q x p h) ]
```


Égalité de John Major

Conor Mc Bride

```
Coq < Inductive JMeq [A:Set;x:A] : (B:Set)B->Prop :=
Coq <         JMeq_refl : (JMeq x x).

Coq < Lemma JMeq_sym : (A,B:Set)(x:A)(y:B)(JMeq x y)->(JMeq y x).

Coq < Axiom JMeq_eq : (A:Set)(x,y:A)(JMeq x y)->(x=y).
```

Définitions coinductives

Christine Paulin

Définitions Coinductives

Une définition inductive ne contient que des objets `bien fondés`. Ceci se reflète dans le principe de récurrence.

Les définitions coinductives servent à modéliser des objets infinis.

On construit ainsi des types contenant des objets infinis ou bien des preuves infinies.

Listes infinies (streams)

```
Coq < Require Streams.

Coq < Print Stream.
CoInductive Stream [A : Set]  : Set :=
      Cons : A->(Stream A)->(Stream A)
For Cons: Position [1] is implicit
```

Remarque

La même définition inductive est vide.

Définition par filtrage

Le filtrage est identique au cas inductif.

Toute valeur de type `(Stream A)` est de la forme `(Cons a l)`.

```
Coq < Print hd.
hd =
[A:Set; x:(Stream A)]Cases x of (Cons a _) => a end
     : (A:Set)(Stream A)->A
Position [1] is implicit

Coq < Print tl.
tl =
[A:Set; x:(Stream A)]Cases x of (Cons _ s) => s end
     : (A:Set)(Stream A)->(Stream A)
Position [1] is implicit
```

Définition par point fixe

Une condition de garde assure la productivité :

```
Coq < Parameter A : Set.
A is assumed

Coq < CoFixpoint repeat : A -> (Stream A)
Coq <   := [a](Cons a (repeat a)).
repeat is corecursively defined

Coq < Parameter f : A -> A.
f is assumed

Coq < CoFixpoint iterate : A->(Stream A)
Coq <   := [a](Cons a (iterate (f a))).
iterate is corecursively defined
```


Définition de fonctionnelles

Combinaison de filtrage et de récurrence.

```
Coq < CoFixpoint map : (Stream A)->(Stream A)
Coq <   := [s]Cases s of (Cons a l)=>(Cons (f a) (map l)) end.
map is corecursively defined
```


Evaluation

```
Coq <  Eval Simpl in [a:A](iterate a).
     = [a:A](iterate a)
     : A->(Stream A)

Coq <  Eval Simpl in [a:A](hd (iterate a)).
     = [a:A]a
     : A->A

Coq <  Eval Simpl in [a:A](hd (tl (iterate a))).
     = [a:A](f a)
     : A->A
```

Les objets de type `Stream` sont des programmes engendrant une infinité de valeurs.

Prédicats coinductifs

On peut également définir des prédicats de manière co-inductive:

```
Coq < CoInductive EqSt : (Stream A) -> (Stream A) -> Prop :=
Coq <  eqst : (s1,s2:(Stream A))
Coq <         (hd s1)=(hd s2)->(EqSt (tl s1) (tl s2))
Coq <         ->(EqSt s1 s2).
EqSt is defined
```

Équivalent à l'existence d'une bisimulation entre `s1` et `s2`.

Preuves plus simples.

Preuve d'égalité

```
Coq < Theorem equality
Coq <   : (x:A)(EqSt (iterate (f x)) (map (iterate x))).
Coq <  Cofix E; Constructor; Simpl.
Coq <  Trivial.
Coq <  Apply E.
Coq <  Save.
```


Un exemple en sémantique

H. Herbelin & C. Paulin

Le langage

Le but de l'exercice est de modéliser un langage de programmation parallèle, composé d'affectations gardées.

On modélise la syntaxe du langage ainsi que sa sémantique.

Une affectation gardée est de la forme

$$garde \rightarrow x_1 := e_1, \ldots, x_n := e_n$$

avec *garde* et e_i des expressions et x_i des variables.

Une expression est soit une variable, soit une constante, soit un symbole de fonction binaire appliqué à deux expressions.

Syntaxe

On se donne de manière abstraite des ensembles arbitraires de constantes, variables et symboles de fonctions.

Construire le type des expressions de manière inductive.

Définir ensuite une clause comme composée d'une expression représentant la garde et d'une liste de couples (variable,expression) représentant les affectations.

On utilisera la bibliothèque `PolyList` sur les listes.

Définir finalement le type des programmes comme étant une liste de clauses.

Sémantique

On interprète toutes les expressions dans le type `nat`.
Il faut se donner une valuation (fonction des variables dans les entiers) ainsi que les interprétations des constantes et des symboles de fonctions.

Définir l'interprétation des expressions (une fonction des valuations dans les valuations) par récurrence sur la structure des expressions.

Définir la sémantique d'une clause comme effectuant le remplacement en parallèle des variables par les valeurs des expressions lorsque la garde n'a pas pour valeur 0.

Sémantique non-déterministe d'un programme

Définir finalement la relation de transition associée au programme, il s'agit d'une relation binaire qui est vérifiée par deux valuations s'il est possible de passer de l'une à l'autre en exécutant une des clauses du programme.

Relations de transition

On s'intéresse maintenant à exprimer des propriétés sur les relations de transition.
On se donne un ensemble `M:Set` arbitraire ainsi qu'une relation binaire `R` sur `M`.

Fermeture transitive Définir la fermeture transitive de la relation `R` de manière inductive.
Établir en particulier que cette relation est transitive, réflexive et contient `R`.

Invariant On dira qu'un prédicat `I` sur `M` est un invariant si `I` est préservé par `R`.
Définir la notion d'invariant.

Relations de transition-suite

Accessibilité On se donne un prédicat `init` sur `M`.
Définir inductivement le predicat `accessible` vrai pour les objets accessibles par la relation `R` à partir d'un objet dans `init`.
Donner une formalisation équivalente utilisant la fermeture transitive de `R`.

Toujours On se donne un prédicat `P` sur `M`.
Définir le prédicat `allways` qui contient les objets qui vérifient `P` et tels que tous les objets atteignables à partir de ce point vérifient `P`.
On pourra donner une définition coinductive et une définition utilisant la fermeture transitive de `R`.