Programmer = démontrer?
La correspondance de Curry-Howard aujourd'hui

Troisième cours

Des armes de construction massive: types inductifs et prédicats inductifs

Xavier Leroy

Collège de France

2018-11-28

COLLÈGE
DE FRANCE
1530

Objectifs du cours

Le cours précédent s'est concentré sur les fonctions et le fragment $\Rightarrow, \forall$ de la logique. Allons plus loin !

Dans les langages de programmation : quels mécanismes pour définir et manipuler des structures de données, au-delà des types de base (nombres, etc) et des fonctions ?

Dans les logiques : comment bien traiter tous les connecteurs : pas juste $\Rightarrow$ et $\forall$, mais aussi $\wedge$, $\vee$, $\bot$, $\exists$? comment définir les objets manipulés par la logique, p.ex. les entiers naturels ? et les prédicats sur ces objets ?

Dans ce cours, nous allons essayer de donner des réponses qui conviennent aux deux questions, dans l'esprit de Curry-Howard !

I

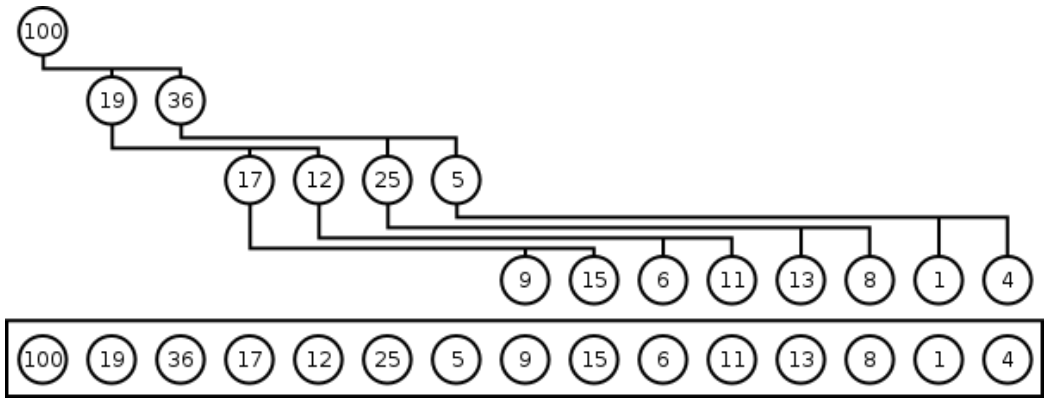
Les structures de données dans les langages de programmation

Types de base et tableaux

Dès FORTRAN et dans presque tous les langages qui ont suivi :

- Des types de base : nombres (entiers, ≪réels≫ (virgule flottante), ≪complexes≫), caractères, ...
- Des tableaux à une ou plusieurs dimensions, indicés par des entiers.

On encode les indirections et les structures chaînées avec des indices de tableaux, p.e.x les arbres tournois (*heaps*) :



Enregistrements (records)

COBOL introduit la notion d'enregistrement (*record*) avec des champs nommés, de types possiblement différents, et possiblement emboîtés.

```
01  Transaction.                          *> enregistrement
       05  Numero      PIC 9(10).         *> champ
       05  Date.                          *> sous-enregistrement
             10  Annee  PIC 9(4).         *> champ
             10  Mois   PIC 99.           *> champ
             10  Jour   PIC 99.           *> champ
```

Adresses, références, pointeurs

Les codes en langage d'assemblage manipulent souvent des adresses en mémoire de diverses données qui ne ≪tiennent≫ pas dans un registre.

Cette pratique entre dans les langages de haut niveau sous le nom de références en ALGOL-W (1966) puis de pointeurs en C, Pascal, etc.

Un pointeur peut être ≪nul≫ (Hoare : *my billion-dollar mistake*).

Enregistrements + pointeurs $\Rightarrow$ codage assez naturel des structures de données chaînées (listes, arbres, …).

```
struct list {
  int head;
  struct list * tail;
};
```

```
struct tree {
  char * name;
  struct tree * left, * right;
}
```

Unions

Si les enregistrements sont des ≪et≫ de plusieurs types, les unions sont des ≪ou≫ de plusieurs types. Apparaissent en Algol 68 :

```
mode node = union (real, int, string);
node n := "1234";
case n in
  (real r):   print(("real:", r)),
  (int i):    print(("int:", i)),
  (string s): print(("string:", s))
  out         print(("?:", n))
esac
```

On a ici une union ≪discriminée≫ : la valeur de type `node` enregistre lequel des 3 cas `real`, `int` et `string` s'applique. Ce n'est pas le cas des unions en C :

```
union u { float f; int i; char * s; };
```

Unions

Pascal combine enregistrements et unions discriminées :

```
type
  exprKind = (VAR, CONST, SUM, PROD);
  expr = record
           (* partie commune *)
           size: integer;
           (* partie variable *)
           case kind : exprKind of
             VAR:        (varName: string);
             CONST:      (constVal: integer);
             SUM, PROD: (left, right: ^expr)
         end
```

Un type universel : les S-expressions

Au lieu de laisser le programmeur définir les structures de données, Lisp offre un type ≪universel≫, les S-expressions, qui représente facilement un grand nombre de structures de données.

$$\begin{aligned} \mathit{sexp} &::= \mathit{atome} \mid (\mathit{sexp}\ \ .\ \ \mathit{sexp}) \\ \mathit{atome} &::= \mathit{nombre} \mid \mathit{symbole} \mid \texttt{nil} \end{aligned}$$

Codages canoniques pour de nombreuses structures de données :

$$\begin{array}{rrcl} \text{Listes :} & (x_1\ \ldots\ x_n) & \equiv & (x_1\ .\ (\cdots(x_n\ .\ \texttt{nil}))) \\ \text{Termes :} & \mathit{cstr}0 & \equiv & \mathit{cstr}0 \\ & \mathit{cstr}(x_1, \ldots, x_n) & \equiv & (\mathit{cstr}\ x_1\ \cdots\ x_n) \\ \text{Fonctions :} & \lambda x.M & \equiv & (\texttt{lambda}\ (x)\ M) \end{array}$$

Un type universel : les termes Prolog

Prolog offre un autre type universel, inspiré des algèbres de termes utilisées en logique et en théorie de la récriture :

$terme ::= nombre \mid atome \mid Variable \mid atome(terme, \ldots, terme)$

Les atomes représentent les constructeurs (constants ou avec arguments) des algèbres de termes.

Produit cartésien et somme binaire

LCF-ML, l'ancêtre des langages de la famille ML, n'offre pas de mécanismes pour déclarer des types enregistrements ni des types union. Il prédéfinit un type ≪produit cartésien≫ et un type ≪somme binaire≫ (union discriminée de 2 types) :

Type :	$t ::= \texttt{int} \mid \texttt{bool} \mid \ldots$		types de base
	$\mid t_1 \to t_2$		types de fonctions
	$\mid t_1 \# t_2$		produit : paires d'un t_1 et d'un t_2
	$\mid t_1 + t_2$		somme : soit un t_1 soit un t_2

NB : par Curry-Howard, le type produit correspond à la conjonction $\wedge$ et le type somme à la disjonction $\vee$.

Récursion en LCF-ML

Un mécanisme d'abstraction de types permet de définir des types récursifs comme des types abstraits équipés de fonctions de construction et de destruction.

Exemple : les arbres binaires portant des valeurs de type $*$ aux feuilles.

```
absrectype * tree = * + * tree # * tree
  with leaf n = abstree(inl n)
   and node (t1, t2) = abstree(inr(t1, t2))
   and isleaf t = isl(reptree t)
   and leafval t = outl(reptree t) ? failwith ‘leafval‘
   and leftchild t = fst(outr(reptree t) ? failwith ‘leftchild‘
   and rightchild t = snd(outr(reptree t) ? failwith ‘leftchild‘
```

Les types algébriques

En 1980 le langage purement fonctionnel HOPE (Burstall, MacQueen, Sannella) introduit la présentation moderne des types algébriques. Repris ensuite par Milner et par Cousineau pour intégration dans CAML et dans Standard ML.

```
type expr =
  | Const of int
  | Infinity
  | Sum of expr * expr
  | Prod of expr * expr
```

Une somme de types produits qui est récursive.

Chaque cas de la somme est discriminé par un constructeur (`Const`, `Infinity`, `Sum`, `Prod`)

Chaque constructeur porte zéro, un ou plusieurs arguments.

Un argument peut être du type en cours de définition (récursion).

Utiliser un type algébrique

Construire des valeurs du type : par application des constructeurs.

```
let e = Sum(Const 1, Prod (Const 2, Infinity))
```

Utiliser des valeurs du type : par filtrage (*pattern matching*).

```
let rec floatval e =
  match e with
  | Const n -> Float.of_int n
  | Infinity -> Float.infinity
  | Sum(e1, e2) -> floatval e1 +. floatval e2
  | Prod(e1, e2) -> floatval e2 *. floatval e2
```

II

Types inductifs

Les types inductifs

Une variante des types algébriques compatible avec la théorie des types.

- En plus : dépendances dans les types des constructeurs et dans les sortes des types inductifs.
- En moins : restrictions sur la récursion pour préserver la normalisation forte et la cohérence logique.

Les types inductifs ont été introduits par Paulin-Mohring et Pfenning en 1989 comme extension du Calcul des Constructions.

Notion primitive en Coq et en Agda, encodée en Isabelle/HOL.

Les types inductifs

Même idée ≪somme de produits + récursion≫ mais autre vision :

univers contenant le type

```
Inductive expr : Type :=
  | Const: nat -> expr
  | Infinity: expr
  | Sum: expr -> expr -> expr
  | Prod: expr -> expr -> expr
```

constructeurs et leur type

Chaque constructeur est une constante de type `expr` ou une fonction de type résultat `expr`.

Le type `expr` est l'ensemble des valeurs engendré par ces constantes et fonctions. (C.à.d. le plus petit ensemble contenant les constantes et stable par les fonctions.)

Des types de données familiers

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

Inductive unit : Type :=
 | tt : unit.

Inductive empty : Type := .    (* type vide, zéro constructeurs *)

Inductive nat : Type :=        (* entiers naturels à la Peano *)
  | O : nat
  | S : nat -> nat.
```

Types inductifs et fonctions

Contrairement aux algèbres de termes usuelles, les types inductifs peuvent contenir des fonctions comme arguments de constructeurs.

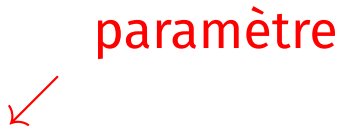
Exemple : les ordinaux de Brouwer.

```
Inductive ord: Type :=
  | Zero: ord
  | Succ: ord -> ord
  | Limit: (nat -> ord) -> ord.
```

Vue comme un arbre, une valeur de type inductif n'est pas nécessairement finie : un noeud peut brancher infiniment en largeur (comme `Limit` ci-dessus). En revanche il n'y a pas de branche infiniment longue.

Types inductifs paramétrés

Un type inductif peut être paramétré par des types (ou des valeurs) et avoir des constructeurs polymorphes.




```
Inductive list (A: Type) : Type :=          (* listes *)
  | nil : list A
  | cons : A -> list A -> list A.

Inductive prod (A: Type) (B: Type) : Type :=
  | pair : A -> B -> prod A B.              (* produit cartésien *)

Inductive sum (A: Type) (B: Type) : Type :=
  | inl : A -> sum A B                      (* somme binaire *)
  | inr : B -> sum A B.
```

Des connecteurs logiques familiers

Par la magie de Curry-Howard, ces types inductifs sont aussi utilisables comme connecteurs logiques : le produit cartésien, c'est la conjonction ; la somme binaire, c'est la disjonction ; etc.

Au lieu d'utiliser les mêmes inductifs dans les types de données et dans les formules logiques, Coq choisit de les définir deux fois, une fois dans l'univers `Prop` et une fois dans l'univers `Type`.

Des connecteurs logiques familiers

```
Inductive and (A: Prop) (B: Prop) : Prop :=  (* «et» logique *)
  | conj : A -> B -> and A B.

Inductive or (A: Prop) (B: Prop) : Prop :=   (* «ou» logique *)
  | or_introl : A -> or A B
  | or_intror : B -> or A B.

Inductive True : Prop :=                     (* vérité *)
  | I : True.

Inductive False : Prop := .                  (* absurdité *)
                                  (* zéro constructeurs! *)
```

`A /\ B` est une notation pour `and A B`

`A \/ B` est une notation pour `or A B`.

Constructeurs à types dépendants

Les constructeurs d'un type peuvent avoir des types dépendants : le type d'un argument peut dépendre de la valeur d'un argument précédent.

C'est le cas pour le type des paires dépendantes, c.à.d. le type $\Sigma x : A.\ B(x)$ de MLTT, mais aussi le quantificateur $\exists x : A.\ P(x)$.

famille de types indexée par `a:A`

```
Inductive sigma (A: Type) (B: A -> Type) : Type :=
  | exist: forall (a: A), B a -> sigma A B.
```

premier argument

second argument (dépendant)

Trois nuances de Sigma

Avec sa distinction `Prop`/`Type`, Coq se retrouve avec trois variantes de Σ-type, toutes trois définies comme des inductifs + des notations :

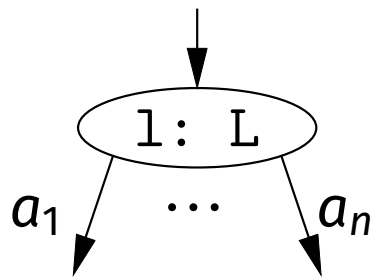
Notation	résultat	`B(x)`	
`exists x:A, B(x)`	`Prop`	`Prop`	quantificateur «il existe»
`{ x: A | B(x) }`	`Type`	`Prop`	type sous-ensemble : un `x:A` avec une preuve que `B(x)`
`{ x: A & B(x) }`	`Type`	`Type`	paire dépendante : un `x:A` et un `B(x)`

Le type W

Un autre composant de MLTT définissable comme type inductif est le type W des arbres bien fondés :

```
Inductive W (L: Type) (A: L -> Type) : Type :=
  | Node: forall (lbl: L), (A lbl -> W L A) -> W L A.
```

`L` est le type des étiquettes portées par chaque noeud. Un noeud étiqueté `l:L` porte n sous-arbres, où n est le nombre d'éléments du type `A l`, chaque sous-arbre étant identifié par un élément de `A l`.



avec $a_1, \ldots, a_n$ les éléments du type `A l`

Le type W

```
Inductive W (L: Type) (A: L -> Type) : Type :=
  | Node: forall (lbl: L), (A lbl -> W L A) -> W L A.
```

Exemples : supposant définis les types `empty`, `unit` et `bool` à 0, 1 et 2 éléments respectivement, voici le type `nat` :

```
W bool (fun b => match b with false => empty
                            | true  => unit   end)
```

Le type des listes de `A` :

```
W (option A) (fun l => match l with None => empty
                                  | Some _ => unit  end)
```

Le type des arbres binaires avec des `A` aux feuilles et des `B` aux noeuds :

```
W (A + B) (fun l => match l with inl _ => empty
                               | inr _ => bool  end)
```

Élimination des types inductifs : par filtrage

Comme pour les types algébriques, une construction de filtrage permet d'éliminer (utiliser en faisant une analyse de cas) un terme de type inductif.

```
Definition not (b: bool) :=
  match b with
  | true => false
  | false => true
  end.
```

```
Definition pred (n: nat) :=
  match n with
  | O => O
  | S p => p
  end.
```

Élimination des types inductifs : par filtrage

Via Curry-Howard, le filtrage correspond à une démonstration par cas.

Exemple : une propriété est vraie pour tout `b: bool` si elle est vraie pour `b = true` et pour `b = false`.

```
Theorem bool_cases: forall (P: bool -> Prop),
                      P true -> P false -> forall b, P b.

Proof fun (P: bool -> Prop) (iftrue: P true) (iffalse: P false)
          (b: bool) =>
        match b with true => iftrue | false => iffalse end.
```

Élimination des types inductifs : par filtrage

La forme générale d'un filtrage pour un inductif à n constructeurs $C_1, \ldots, C_n$ d'arité $a_1, \ldots, a_n$ est :

$$
\begin{array}{l}
\texttt{match } e \texttt{ with} \\
\mid\ C_1\ x_1^1\ \ldots\ x_1^{a_1}\ \texttt{=>}\ b_1 \\
\mid \ldots \\
\mid\ C_n\ x_n^1\ \ldots\ x_n^{a_n}\ \texttt{=>}\ b_n \\
\texttt{end}
\end{array}
$$

La règle de réduction correspondante :

$$\texttt{match } C_i\ e^1\ \ldots\ e^{a_i}\ \texttt{with}\ \ldots\ \texttt{end} \quad \to \quad b_i\{x_i^1 \leftarrow e^1, \ldots, x_i^{a_i} \leftarrow e^{a_i}\}$$

Exemple des entiers naturels :

```
match O   with O => a | S p => b end  → a
match S n with O => a | S p => b end  → b{p←n}
```

Élimination des types inductifs : par récurseurs

Une alternative à la construction de filtrage (`match`) est que chaque définition `Inductive` définisse une fonction de récursion sur le type, qui combine analyse de cas et calcul récursif.

Pour un type T à n constructeurs, ce récurseur est de la forme

$$T\texttt{\_rec}\ \mathit{valeur}\ \mathit{cas}_1\ \ldots\ \mathit{cas}_n$$

Par exemple, pour les booléens, les options, et les naturels :

```
bool_rec true  a b  →  a
bool_rec false a b  →  b

option_rec None     a b  →  a
option_rec (Some x) a b  →  b x

nat_rec O     a b  →  a
nat_rec (S n) a b  →  b (nat_rec n a b)  (* <- récursion *)
```

Itération

Les récurseurs fournissent une forme simple de récursion appelée itération, où les appels récursifs se font uniquement et toujours sur les arguments immédiats des constructeurs récursifs.

Par exemple, la fonction ≪fois deux≫ sur les entiers de Peano est une itération, mais pas la fonction de Fibonacci :

```
let rec double n =      (* nat_rec n O (fun x => S (S x)) *)
  match n with O -> O | S p -> S (S (double p))

let rec fib n =
  match n with O -> S O
             | S p -> match p with O -> S O
                                 | S q -> fib p + fib q
```

Exercice : définir par itération la fonction `fib' n` $\overset{def}{=}$ `(fib n, fib (S n))`.

Filtrage et récursion

Pour opérer sur des types inductifs, le filtrage (`match`) doit être combiné avec un mécanisme pour définir des fonctions récursives (`let rec` en Caml, `Fixpoint` en Coq).

```
Fixpoint double (n: nat) : nat :=
  match n with O => O | S p => S (S (double p)) end.

Fixpoint fib (n: nat) : nat :=
  match n with O => S O
             | S p => match p with O => S O
                                 | S q => fib p + fib q end
  end.
```

Une condition de garde garantit la terminaison. Typiquement, les appels récursifs doivent se faire sur un sous-terme strict de l'argument (récursion structurelle).

Terminaison et cohérence

Presque toujours, si on peut définir une expression qui se réduit à l'infini, on peut lui donner un type vide (`False` ou `empty`), qui permet de ≪démontrer≫ n'importe quelle proposition `P`.

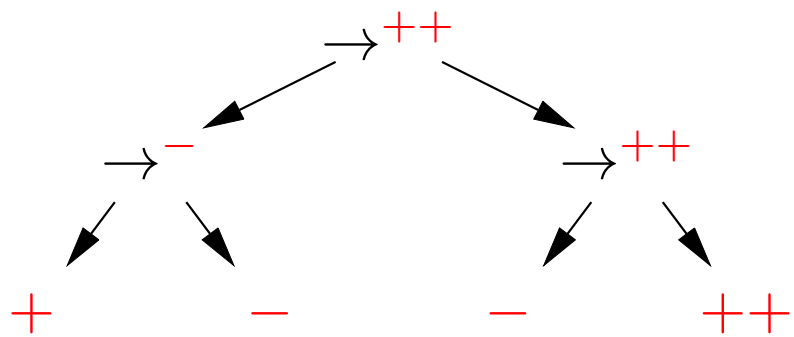
✘
```
Fixpoint boucle (n: nat) : False := boucle (S n).
```

✘
```
Theorem incohérence (P: Prop) : P :=
    match boucle 0 with end.
```

On ne peut donc pas avoir de récursion générale dans un langage comme Coq ou Agda.

Positivité

Pour les mêmes raisons, un type inductif doit apparaître en position strictement positive dans les types des arguments de ses constructeurs.



$++$	strictement positive	à gauche de zéro flèches
$+$	positive	à gauche de $2n$ flèches
$-$	négative	à gauche de $2n+1$ flèches

Positivité

Un prédicat inductif en occurrence négative nous donne immédiatement une contradition :

✘
```
Inductive P : Prop := Pintro : (P -> False) -> P.

Lemma paradox: P <-> ~P.
```

Un type inductif en occurrence négative nous donne le lambda-calcul pur, et donc des expressions qui divergent :

✘
```
Inductive lam : Type := Lam: (lam -> lam) -> lam.

Definition app (a b: lam) : lam := match a with Lam f => f b end.
Definition delta : lam := Lam (fun x => app x x).
Definition omega : lam := app delta delta.   (* diverge! *)
```

Positivité

Un type inductif en occurrence «doublement négative» (positive mais pas strictement) donne un paradoxe.

✘
```
Inductive A : Type := Aintro : ((A -> Prop) -> Prop) -> A.
```

```
Definition f (x: A -> Prop) : A := Aintro (fun y => x = y).
```

On montre que `f` est une injection de `A -> Prop` (= les parties de `A`) dans `A`, ce qui est impossible par «cardinalité».

Une diagonalisation à la Cantor produit une contradiction.

(Coquand, *A new paradox in type theory*, Studies in Logic and the Foundations of Mathematics 134, 1995).

III

Familles inductives

Sur quoi porte la récursion ?

Type inductif paramétré :

- Fonction des paramètres vers un type récursif.
- Les paramètres ne changent pas dans les types des constructeurs.

```
Inductive list (A:Type): Type :=
| nil: list A
| cons: A -> list A -> list A.
```

```
(* même paramètre *)
```

Famille inductive :

- Fonction récursive des paramètres vers un type.
- Les paramètres peuvent prendre différentes valeurs dans les types des constructeurs.

```
Inductive t: Type -> Type :=
  | A: t nat
  | B: t bool.
```

```
(* différents paramètres *)
```

Exemples de familles inductives

Famille inductive : le type `fin n` des entiers entre 0 et `n`.

```
Inductive fin: nat -> Type :=
  | Zero : forall (n: nat), fin n
  | Succ : forall (n: nat), fin n -> fin (S n).
```

Famille inductive avec un paramètre :
le type `vec A n` des listes de `A` de longueur `n`.

```
Inductive vec (A: Type): nat -> Type :=
  | nil: vec A 0
  | cons: forall (n: nat), A -> vec A n -> vec A (S n).
```

Exercice : définir une fonction sûre d'accès au n-ième élément

```
nth: forall (A: Type) (n: nat),  vec A (S n) -> fin n -> A.
```

Prédicats inductifs

Les familles inductives sont très utiles pour définir des prédicats par un système de règles d'inférence.

- Prédicat P à n arguments de types $A_1, \ldots A_n$
  $\Rightarrow$ `Inductive` $P : A_1 \to \cdots \to A_n \to$ `Prop`
- Axiome $\Rightarrow$ constructeur constant
- Règle d'inférence à k prémisses $\Rightarrow$ constructeur à k arguments.

La récursion structurelle sur P fournit un puissant principe de preuve par récurrence sur la structure d'une dérivation et par cas sur la dernière règle utilisée.

Exemple : le prédicat «être pair» sur les entiers

$$\frac{\texttt{even}(0) \qquad \texttt{even}(n)}{\texttt{even}(S(S(n)))}$$

```
Inductive even: nat -> Prop :=
  | even_0:
      even 0
  | even_S: forall n,
      even n ->
      even (S (S n)).
```

Exemple : représenter une logique

Les règles d'inférence :

$$\Gamma \vdash \top \ (\top\mathsf{I}) \qquad \Gamma_1, P, \Gamma_2 \vdash P \ (\mathsf{Ax})$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\ (\Rightarrow\mathsf{I}) \qquad \frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\ (\Rightarrow\mathsf{E}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ (\bot\mathsf{E})$$

La représentation des formules logiques :

```
Inductive formule : Type :=
  | Vrai: formule                                  (* ⊤, vérité *)
  | Faux: formule                                  (* ⊥, absurdité *)
  | Imp : formule -> formule -> formule.           (* ⇒, implication *)
```

Exemple : représenter une logique

La transcription des règles de déduction :

```
Inductive sequent : list formule -> formule -> Prop :=
  | VraiI: forall G,
      sequent G Vrai
  | Ax: forall G1 P G2,
      sequent (G1 ++ P :: G2) P
  | ImpI: forall G P Q,
      sequent (P :: G) Q ->
      sequent G (Imp P Q)
  | ImpE: forall G P Q,
      sequent G (Imp P Q) -> sequent G P ->
      sequent G Q
  | FauxE: forall G P,
      sequent G Faux ->
      sequent G P.
```

$$\Gamma \vdash \top$$

$$\Gamma_1, P, \Gamma_2 \vdash P$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}$$

$$\frac{\Gamma \vdash P \Rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}$$

Les ordres bien fondés

Un ordre est bien fondé s'il n'existe pas de suite infinie strictement décroissante.

Une caractérisation plus positive : tout `x` est ≪accessible≫, i.e. toutes les suites décroissantes issues de `x` sont finies.

```
Inductive Acc (A: Type) (ord: A -> A -> Prop) : A -> Prop :=
  | Acc_intro: forall x:A,
      (forall y:A, ord y x -> Acc A ord y) -> Acc A ord x.

Definition well_founded (A: Type) (ord: A -> A -> Prop) :=
  forall x:A, Acc A ord x.
```

Une récurrence structurelle sur une preuve de `Acc x` correspond à une récurrence Noetherienne sur `x`.

L'égalité comme un prédicat inductif

En MLTT comme dans beaucoup de logiques, l'égalité est une construction primitive, avec ses règles de raisonnement spécifiques.

Elle peut aussi être définie comme un prédicat inductif :

```
Inductive equal (A: Type): A -> A -> Prop :=
  | reflexivity: forall (x: A), equal x x.
```

Pour montrer une propriété `P x y` sous l'hypothèse `H: equal A x y`, il suffit de faire une analyse de cas sur `H`. Le seul cas possible étant `H = reflexivity z`, il suffit de montrer `P z z` pour tout `z`.

(Plus de détails dans le cours du 23 janvier ≪Qu'est-ce que l'égalité ?≫.)

IV

Les types algébriques généralisés (GADT)

Les GADT

Generalized Algebraic Data Types = types algébriques généralisés

C'est l'équivalent des familles inductives pour des langages de programmation comme OCaml et Haskell, qui n'ont pas de types dépendants (dans toute leur généralité).

Ce sont des types algébriques paramétrés (`'a ty`) où les constructeurs ne renvoient pas tous `'a ty` mais peuvent renvoyer des instances τ `ty`.

Exemple : une représentation optimisée des tableaux en OCaml.

```
type 'a compact_array =
| Array: 'a array -> 'a compact_array    (* cas par défaut *)
| Bytes: bytes    -> char compact_array  (* cas optimisé *)
| Bools: bitvect  -> bool compact_array  (* cas optimisé *)
```

Historique des GADTs

1992 Coquand : *Pattern-matching with dependent types*. Combiner familles inductives et filtrage `match`.

1992 Läufer : *Polymorphic Type Inference and Abstract Data Types*. Les ≪types existentiels≫, un cas particulier de GADT.

1994 Augustsson, Petersson : *Silly type families* (manuscrit). Lever la restriction de régularité des types des constructeurs. Problèmes pour inférer les types des `match`.

2003 Xi, Chen, Chen : *Guarded Recursive Datatype Constructors*. Redécouverte des mêmes idées.

2006 Peyton-Jones et al + Pottier et Régis-Gianas. Premiers algorithmes d'inférence partielle de types pour les filtrages sur les GADTs.

2007 GHC 6.8.1 : introduction des GADTs dans Haskell.

2012 OCaml 4.00 : introduction des GADTs dans Caml.

Des valeurs qui déterminent des types

Un cas particulier de types dépendants qui se code bien avec des GADTs.

Exemple : la fonction `sprintf` (impression formattée)

```
sprintf "toto" : string
sprintf "var = %d" : int -> string
sprintf "%s = %d" : string -> int -> string
```

On obtient une version typée `sprintf: 'a format -> 'a`
en codant les formats avec le GADT `'a format` suivant :

```
type _ format =
  | Lit: string * 'a format -> 'a format
  | Param_int: 'a format -> (int -> 'a) format
  | Param_string: 'a format -> (string -> 'a) format
  | End: string format

sprintf (Lit("toto", End)) : string
sprintf (Lit("var = ", Param_int End)) : int -> string
sprintf (Param_string (Param_int End)) : string -> int -> string
```

Types existentiels et égalités de types

Les GADTs permettent de quantifier existentiellement sur un type :

```
type printable =
  | Printable: 'a * ('a -> string) -> printable
```

Se lit comme $\exists A : \texttt{Type}.\ A \times (A \to \texttt{string})$.

On peut aussi définir l'égalité entre deux types :

```
type ('a, 'b) equal =
  | Equal: ('c, 'c) equal
```

Réciproquement, tout GADT peut s'écrire comme un type algébrique paramétré usuel + des types existentiels + des égalités de types.

V

Codages fonctionnels

Types inductifs : notion primitive ou dérivée ?

Les types inductifs sont une notion primitive en Coq et Agda, de même que les type W dans la théorie des types de Martin-Löf.

Mais il s'en faut de peu pour qu'on puisse les définir dans les Pure Type Systems du cours précédent, avec seulement des lambda-termes et des Pi-types.

Codage non typé

Une généralisation du codage de Church des entiers dans les lambda-termes purs. On part des équations définissant le récurseur d'un type inductif :

$$
\begin{array}{lllll}
\texttt{bool\_rec true} & a\ b & \to & a \\
\texttt{bool\_rec false} & a\ b & \to & b \\
\texttt{nat\_rec O} & a\ b & \to & a \\
\texttt{nat\_rec (S } n\texttt{)} & a\ b & \to & b\ \texttt{(nat\_rec } n\ a\ b\texttt{)}
\end{array}
$$

On dit que les récurseurs sont la fonction identité, et donc constructeur = récurseur :

$$
\begin{array}{lllll}
\texttt{true} & a\ b & \to & a \\
\texttt{false} & a\ b & \to & b
\end{array}
\qquad\qquad
\begin{array}{lllll}
\texttt{O} & a\ b & \to & a \\
\texttt{(S } n\texttt{)} & a\ b & \to & b\ \texttt{(}n\ a\ b\texttt{)}
\end{array}
$$

On en déduit les lambda-termes qui correspondent aux constructeurs :

$$
\begin{array}{l}
\texttt{true} = \lambda a.\ \lambda b.\ a \\
\texttt{false} = \lambda a.\ \lambda b.\ b
\end{array}
\qquad\qquad
\begin{array}{l}
\texttt{O} = \lambda a.\ \lambda b.\ a \\
\texttt{S} = \lambda n.\ \lambda a.\ \lambda b.\ b\ (n\ a\ b)
\end{array}
$$

Codage simplement typé

On est forcé de donner des types monomorphes non seulement aux constructeurs mais aussi aux filtrages ! Par exemple pour les Booléens :

```
type bool = t → t → t      (* pour un type t fixé *)
true : bool = λa : t. λb : t. a
false : bool = λa : t. λb : t. b
```

On peut faire un ≪if then else≫ dont le résultat est de type t mais d'aucun autre type. De même pour les entiers : en prenant

```
type nat = t → (t → t) → t (* pour un type t fixé *)
```

on peut définir l'addition de deux entiers, ainsi que la multiplication, mais pas l'exponentielle p.ex.

Théorème (Schwichtenberg)

Les fonctions $\mathbb{N} \to \mathbb{N}$ définissables en lambda-calcul simplement typé sont les polynômes étendus (par un test à zéro).

Codage dans système F

Avec système F, on peut quantifier universellement sur le type du résultat du filtrage :

$$\texttt{type bool} = \forall X.\ X \to X \to X$$

$$\texttt{type nat} = \forall X.\ X \to (X \to X) \to X$$

Ceci rend l'approche ≪codage≫ très expressive, en pratique et même en théorie.

Théorème (Girard)

Toutes les fonctions $\mathbb{N} \to \mathbb{N}$ prouvablement totales dans l'arithmétique de Peano du second ordre sont définissables dans système F.

Codage dans système F_ω

Le codage du type inductif s'obtient à partir des types des constructeurs où on remplace le type inductif par une variable de type R (= type du résultat) quantifiée universellement. Les paramètres deviennent des λ de F_ω.

```
pair: A -> B -> prod A B
                prod = λA : *. λB : *. ∀R : *. (A -> B -> R) -> R

inl: A -> sum A B
inr: B -> sum A B
                sum = λA : *. λB : *. ∀R : *. (A -> R) -> (B -> R) -> R

nil: list A
cons: A -> list A -> list A
                list = λA : *. ∀R : *. R -> (A -> R -> R) -> R
tt: unit
                unit  = ∀R : *. R -> R
                empty = ∀R : *. R
```

Codage dans système F_ω

Le résultat s'étend aux GADTs et aux familles inductives, en remplaçant le constructeur de type par une variable de la même sorte, quantifiée universellement.

```
Array: ∀X. array X -> compact_array X
Bytes: bytes -> compact_array char
Bools: bitvect -> compact_array bool
       compact_array = λA : ∗. ∀R : ∗ ⇒ ∗.
                       (∀X. array X -> R X) ->
                       (bytes -> R char) ->
                       (bitvect -> R bool) -> R A

refl: ∀X. equal X X
       equal = λA : ∗. λB : ∗. ∀R : ∗ ⇒ ∗ ⇒ ∗. (∀X. R X X) -> R A B.
```

Limitation : pas d'élimination dépendante

En général, le type d'une analyse de cas peut dépendre de la valeur analysée. C'est particulièrement vrai quand le type est une proposition et l'analyse de cas sa démonstration.

Exemple : pour montrer $\forall b : \texttt{bool}.\, P(b)$, on fait une analyse de cas
`match b with false => ... | true => ... end`
dont le type est `P b`.

L'éliminateur n'est donc plus de la forme $\texttt{bool} \to R \to R \to R$
mais de la forme $\texttt{bool} \to R\ \texttt{false} \to R\ \texttt{true} \to \forall b : \texttt{bool}, R\ b$.

Un codage fonctionnel mènerait à un type circulaire…

```
bool = ∀R: bool -> Type.  R -> R -> ∀b: bool. R b
```

Limitation : pas de ≪grande élimination≫

Un filtrage peut produire un résultat dans un univers plus ≪haut≫ que celui de la valeur filtrée, p.ex. on construit un type par filtrage sur un `nat`.

```
Fixpoint vec (A: Type) (n: nat) : Type :=
    match n with
    | O => unit
    | S p => prod A (vec A p)
    end.
```

Dans un codage fonctionnel cela mène à une incohérence d'univers.

VI

Pour aller plus loin

Les limites de la condition de garde

```
Fixpoint f (x: T) :=  ...  f a ...  f b ...
```

La condition de garde ≪`a`, `b` sous-termes stricts de `x`≫ est

1. pas toujours bien définie (nombreuses variantes en Coq);
2. peu compatible avec l'abstraction.

En particulier, ne traite pas les cas de récursion ≪enveloppée≫ et de récursion à travers d'autres fonctions :

```
Fixpoint f (x: T) :=  ...  f (g x) ...
```

```
Fixpoint f (x: T) :=  ...  f ( ... f x ... ) ...
```

Exemple de récursion non structurelle

La division euclidienne sur les entiers de Peano :

✔
```
Fixpoint minus (p q: nat) :=
  match p, q with
  | O, _ => O
  | _, O => q
  | S p', S q' => minus p' q'
  end.
```

✘
```
Fixpoint div (p q: nat) :=    (* quotient de p par q+1 *)
  match p with
  | O => O
  | S p' => S (div (minus p' q) q)
  end.
```

`div` termine parce que `minus p' q <= p'`. Une récursion Noetherienne (sur l'ordre bien fondé des entiers) est nécessaire.

Les types avec tailles (*sized types*)

Pour chaque type inductif T on distingue :

- T^i : les $t : T$ de taille $< i$. (i entier ou ordinal)
- T^{i+1} : les $t : T$ de taille $\leq i$.

On peut alors typer la récursion comme suit :

$$\frac{\Gamma, f : T^i \to S \vdash e : T^{i+1} \to S}{\Gamma \vdash \texttt{fixpoint}\ f = e : T \to S}$$

Plus le sous-typage $T^i <: T^j$ si $i < j$.

Dans l'exemple précédent, on peut montrer $\texttt{minus} : \texttt{nat}^i \to \texttt{nat} \to \texttt{nat}^i$, et conclure que `div` est une récursion correcte.

Les types avec tailles (*sized types*)

Agda utilise les types dépendants pour ajouter une annotation de taille sur un type inductif :

```
sized type SNat : Size -> Set where
  zero: ∀(i: size) -> SNat ($ i)
  succ: ∀(i: size) -> SNat i -> SNat ($ i)
```

`SNat i` est le type des entiers de taille $\leq$ `i`.
`Size` est un type prédéfini $\approx$ ordinaux.
`$` est $\ll$taille + 1$\gg$.

Autres formes de types inductifs

Inductifs mutuels :

```
Inductive tree : Type :=
| Leaf: A -> tree
| Node: forest -> tree
and forest : Type :=
| Nil: forest
| Cons: tree -> forest -> forest.
```

Inductifs emboîtés :

```
Inductive tree : Type :=
| Leaf: A -> tree
| Node: list tree -> tree.
```

OK en Coq et Agda. Principes de récurrence parfois trop faibles.

Autres formes de types inductifs

Induction-récursion : (P. Dybjer)

- Un type inductif `A : Type`
- qui utilise une fonction `A -> B`.

Induction-induction :

- Un type inductif `A : Type`
- qui utilise une famille inductive `B : A -> Type`.

Motivation : modéliser une théorie des types (p.ex. MLTT) dans une autre.
(J. Chapman, *Type theory should eat itself*, 2009.)

Les types quotients

Une construction très utilisée en mathématiques :
le quotient d'un ensemble par une relation d'équivalence.

Exemple : $\mathbb{Q} = (\mathbb{Z} \times \mathbb{Z}^*) \,/\, R$ avec $R\,(p,q)\,(p',q') \overset{def}{=} pq' = p'q$.

En théorie des types, pas de bonne notion générale de ≪type quotient≫.

Dans des cas particuliers on peut cependant définir le type quotient comme le type sous-ensemble des représentants canoniques :

```
Definition Q_canon (pq: Z * Z) : Prop :=
  let (p, q) := pq in q > 0 /\ gcd p q = 1.

Definition Q = { pq: Z * Z | Q_canon pq }
```

Les types inductifs ≪supérieurs≫ (HIT)

(Higher Inductive Types)

Un concept de la théorie homotopique des types : définir un type inductif par ses constructeurs et les égalités qu'ils satisfont.

```
Inductive Z2Z : Type :=
| O: Z2Z
| S: Z2Z -> Z2Z
| mod2: S (S O) = O.
```

Garantir que les filtrages sont compatibles avec ces égalités.

```
match (n: Z2Z) with
| O => a
| S p => b p
| mod2 => (* preuve que a = b (S O) *)
end.
```

(À suivre dans le cours du 23 janvier ≪Qu'est-ce que l'égalité ?≫.)

VII

Bibliographie

Bibliographie

- Yves Bertot et Pierre Castéran. *Le Coq'Art*, `http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`, traduction française du livre *Interactive Theorem Proving and Program Development*. Chapitres 6, 8, 14, 15.

- Adam Chlipala. *Certified Programming with Dependent Types*. `http://adam.chlipala.net/cpdt/`. Chapitres 3, 4, 6, 8, 9.