# Assistants de Preuve

Bruno Barras et Christine Paulin-Mohring

Université Paris-Sud 11, INRIA

25/11/08

# Plan

# Objectifs du cours

- Etude des assistants de preuve interactifs s'appuyant sur des théories des types d'ordre supérieur, en particulier l'assistant de preuves Coq.
  - Comment construire un environnement pour le développement de preuves formelles sur machine
- Etude des définitions inductives
  - Théorie et pratique
- Application à la preuve de programmes.
  - Programmation fonctionnelle avec types dépendants
  - Programmes impératifs

# Déroulement du cours

- Deux heures de cours + 1 heure de TD sur machine (salle 1C22)
- Plan
  - 25/11 - CP - Définitions inductives 1
  - 02/12 - CP - Définitions inductives 2
  - 09/12 - BB - Programmation fonctionnelle 1
  - 16/12 - BB - Programmation fonctionnelle 2
  - 06/01 - BB - Architecture d'un système de preuve
  - 13/01 - BB - Modèles, réalisabilité
  - 20/01 - CP - Types dépendants, extraction
  - 27/01 - CP - Preuve de programmes impératifs
  - 10/02 - Examen

# Evaluation

- Un examen classique.
- Un projet facultatif qui peut compter pour la moitié de la note finale.
- Le projet est réalisé en Coq et donne lieu à la rédaction d'un rapport court et à une soutenance (10-15mn).

# Informations pratiques

- Page WEB du cours (polycopié, transparents, TDs et correction, anciens projets et examens) :
  `http://logical.inria.fr/mpri`
- Bruno Barras, projet Typical, INRIA Saclay - Île-de-France et LIX, Ecole Polytechnique
  Bruno.Barras@inria.fr
- Christine Paulin, projet PROVAL, INRIA Saclay - Île-de-France et LRI, Université Paris-Sud
  Christine.Paulin@lri.fr

# Preuves sur machine

Pour faire des preuves sur machine on a besoin :

- D'une langage de représentation d'objets : entiers, fonctions, ...
- D'un langage pour représenter des propriétés des objets : logique du premier ordre, d'ordre supérieur.
- D'une méthode de construction/vérification de preuve.

Approche basées sur la logique d'ordre supérieur :

- un lambda-calcul typé pour la représentation des objets et des propriétés
  $\neq$ théorie des ensembles (premier ordre)
- des tactiques ou des termes de preuve bien typés pour construire et vérifier des preuves.

# Des exemples d'applications

- Formalisation de sémantiques de langages comme JavaCard, certification de fonctions de sécurité (Gemplus, Trusted Logic)
- Preuve du théorème des 4 couleurs (G. Gonthier, B. Werner - INRIA - Microsoft)
- Développement d'un compilateur C optimisant certifié (Compcert, X. Leroy)
- Formalisation et raisonnement sur des calculs flottants (S. Boldo, ...)
- Développement d'analyseurs statiques certifiés (D. Pichardie)
- ...

# Un peu d'histoire

- Calcul des Constructions (Coquand-Huet)
  - Une seule sorte **Prop** pour représenter les types et les propositions
  - Abstraction/Application/Produit comme seuls opérateurs
  - Tous les produits possibles : polymorphisme $\forall A : \mathbf{Prop}, A \to A$, types dépendants $P : A \to \mathbf{Prop}$
  - Représentation des données et des propriétés par des codages imprédicatifs.
- Ajout d'une hiérarchie d'univers.
  extension du polymorphisme : $A :$ **Type** peut être instancié par **Prop**, $A \to$ **Prop** **Prop** $\to$ **Prop** ...)
- Ajout d'une distinction **Prop**, **Set** entre propriété logique et propriété calculatoire.
- Théorie des types de Martin-Löf : pas d'imprédicativité mais des constructions de base définies par des règles de construction et d'élimination.
- Le Calcul des Constructions Inductives.
  une tentative de fusion de ces deux formalismes :
  - Des définitions (co)-inductives primitives pour leur facilité d'utilisation (moins de codage) et leur généralité (calculatoire et logique).
  - Une logique d'ordre supérieur pour l'expressivité.

# La structure du Calcul des Constructions Inductives

Calcul des Constructions Inductives (version prédicative – Coq $\geq$ 8.0)

=

Calcul des Constructions sur **Prop** et **Type**

*pour la logique d'ordre supérieur*

*pour les types imprédicatifs (historiquement)*

+

hiérarchie de type **Set** : **Type**= $\mathbf{Type}_1$ : $\mathbf{Type}_2$ : $\mathbf{Type}_3$ ...

*pour l'extraction de programme*

*pour l'expressivité logique*

+

types inductifs

*pour des formalisations et des types de données plus « naturels »*

*pour un surcroît d'expressivité calculatoire*

*pour un surcroît d'expressivité logique*

# Le Système $F$ vu comme logique propositionnelle

système $F$ “propositionnel”

| | |
|---|---|
| $\Pi A : \textbf{Prop}.\ B$ | $\forall A : \textbf{Prop}, B$ |
| $A \to B$ | $A \to B$ |
| $\Pi C : \textbf{Prop}.\ (A \to B \to C) \to C$ | la conjonction $A \wedge B$ |

et l’abstraction, l’application, et les variables implémentent les règles d’inférence de la logique

# Le Système $F$ vu comme calcul

système $F$ “calculatoire”

| | |
|---|---|
| $\Pi A : \mathbf{Prop}.\ B$ | $\forall A : \mathbf{Set}, B$ |
| $A \to B$ | $A \to B$ |
| $\Pi C : \mathbf{Prop}.\ (A \to B \to C) \to C$ | produit $A \times B$ |
| $\lambda A : \mathbf{Prop}.\ t$ | **fun**(A :**Set**) $\Rightarrow$t |
| $\lambda x : A.\ t$ | **fun**(x :A) $\Rightarrow$t |
| $t\ A$ | $t\ A$ |
| $t\ u$ | $t\ u$ |
| $x$ | $x$ |
| $\lambda C : \mathbf{Prop}.\lambda f : A \to B \to C.f\ a\ b$ | la paire $(a, b)$ |

# Du Système $F$ au Calcul des Constructions

- Intérêt : pouvoir parler de la couche calculatoire du système $F$ à l'intérieur de la couche logique du système.
- Ajout des produits de la forme

$$A \to \mathbf{Prop}$$

- ... et ajout du polymorphisme d'ordre supérieur, produits de la forme

$$(\mathbf{Prop} \to \mathbf{Prop}) \to \mathbf{Prop}$$

- $\hookrightarrow$ Le Calcul des Constructions fait de la correspondance de Curry-Howard-de Bruijn une identité.
- $\hookrightarrow$ Une logique originale capable de "parler" des preuves.

# Le Calcul des Constructions : un système “complexe”

*Les couches calculatoires et logiques sont superposées*

$\hookrightarrow$ introduction de **Set** au côté de **Prop** duplication de la couche système $F$ avec cette intention :

- dans **Prop**, la nature des preuves n'importe pas ; le principe d'indiscernabilité ($\forall P : Prop \forall pq : P.\, p = q$) est admissible.
- dans **Set**, les objets sont discriminables ; par exemple, dans le type des booléens, `true` $\neq$ `false` sera admissible
- **Prop** peut être interprété par un type booléen : toute proposition prouvable est interprété par `true` et toute proposition prouvablement fausse par `false`.
- bien qu'on puisse coder les entiers, on ne peut pas prouver $0 \neq 1$

# Système $U$

et si le niveau haut de $F$ était polymorphe imprédicatif

- Ajout de **Type** avec **Prop** : **Type** et

$$\frac{A : \textbf{Prop}}{\Pi K : \textbf{Type}.\ A : \textbf{Prop}}$$

$$\Pi K : \textbf{Type}.\ K : \textbf{Type}$$

- ... on obtient un système prouvablement incohérent :
  - encodage du paradoxe de Burali-Forti (Girard 1978),
  - du paradoxe de Russell (Miquel 2000),
  - ou même d'un quasi-point-fixe (Hurkens).
- pouvoir raisonner sur les preuves d'un système imprédicatif de prédicats ... est incohérent

# Système $F_{\omega.2}$

et si le niveau haut de $F_\omega$ était simplement polymorphe mais prédicatif

- L'ajout de $\mathbf{Type}_2$ au dessus de $\mathbf{Type}_2$ permet de rendre le niveau haut du système $F_\omega$ polymorphe sans être imprédicatif

$$\Pi K : \mathbf{Type}.\ A : \mathbf{Prop}$$

$$\Pi K : \mathbf{Type}.\ K : \mathbf{Type}_2$$

- ... élève la force logique au niveau de la théorie des ensembles de Zermelo
- En particulier : on peut y définir un type des entiers qui vérifie $0 \neq 1$.
- Suite logique, une hiérarchie d'univers

$$\mathbf{Type}_1 : \mathbf{Type}_2 : \mathbf{Type}_3 \ldots$$

- ... en ajoutant la dépendance en les preuves, on obtient le calcul des constructions étendu avec univers.

# Les inconvénients du codage polymorphe des définitions inductives

*Cas du codage imprédicatif*

- $0 \neq 1$ n'est pas dérivable
- l'induction n'est pas « directement » dérivable (on n'a que le récurseur)
- *Cas du codage prédicatif dans le calcul avec univers*
  - OK au niveau expressivité (on a $0 \neq 1$ et une induction « indirecte »)
  - *Mais* pas de prédécesseur en une étape
  - pas "naturel"
  - difficile d'écrire des outils automatiques capables de distinguer entre les constructeurs de types inductifs et termes arbitraires
- Mise en place de types inductifs primitifs à la Martin-Löf

# Le Calcul des Constructions Inductives (Coq $\geq$ 5.6)

*Un schéma général de constructions de types inductifs*

- critère de positivité (pour garantir l'existence d'un plus petit ensemble contenant un ensemble donné de constructeurs)
- décomposition des récurseurs en un opérateur d'analyse de cas (`match-with`) et un combinateur pur de point fixe (`fix`)
- critères de terminaison des points fixes
- *Des conditions spécifiques d'élimination selon les sortes*
  - respecte le côté calculatoire de **Set** et **Type** et le côté purement logique de **Prop**
  - évite les paradoxes liés à l'imprédicativité
- *Quelques conséquences*
  - $0 \neq 1$ dérivable
  - principe d'induction dérivable
  - axiome du choix intuitionniste dérivable

# Les limites du Calcul des Constructions Inductives

- L'imprédicativité de la sorte calculatoire **Set** donne au Calcul des Constructions Inductives (CCI) un côté fortement intuitionniste (existence de modèles calculatoires uniquement)
- axiome du choix avec logique classique incohérents, extensionnalité des fonctions pas validée
- limite les possibilités de formalisation de mathématiques classiques
- Choix : passage à un CCI avec **Set** prédicatif

# Le Calcul des Constructions Inductives prédicatif (Coq $\geq$ 8.0)

- La sorte **Set** rejoint la hiérarchie des types (**Set**= $\mathbf{Type}_0$)
- Plus de différence (sauf historique) entre mettre les types de donnée dans **Set** ou dans **Type**.
- Une approche qui rejoint celle de HOL (mais avec types inductifs et hiérarchie d'univers)
- Compatible avec les axiomes mathématiques standard : logique classique, axiome du choix classique, extensionnalité (justifiés par plongement dans la théorie des ensembles)

# Types inductifs : l'exemple des booléens

*en Objective Caml*

```
type bool = | true | false
```

# Types inductifs : l'exemple des booléens

*en Objective Caml*

```
type bool = | true | false
```

*dans la couche Système F de Coq*

```
Definition bool := ∀P:Prop, P → P → P.
Definition true  : bool := fun P:Prop ⇒ fun H1 H2 ⇒ H1.
Definition false : bool := fun P:Prop ⇒ fun H1 H2 ⇒ H1.
```

# Types inductifs : l'exemple des booléens

*en Objective Caml*

```
type bool = | true | false
```

*dans la couche Système F de Coq*

```
Definition bool := ∀P:Prop, P → P → P.
Definition true  : bool := fun P:Prop ⇒ fun H1 H2 ⇒ H1.
Definition false : bool := fun P:Prop ⇒ fun H1 H2 ⇒ H1.
```

*comme type inductif primitif du Calcul des Constructions Inductives*

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

# Types inductifs : exemples

*en Objective Caml*

```
type bool = | true | false
let andb b1 b2 =  match b1 with
    | true -> b2 | false -> false
type nat = | O | S of nat
let rec fact n =  match n with
    | O -> S(O) | S(p) -> n * fact p
```

*dans le CCI, syntaxe Coq*

```
Inductive bool : Type
| true : bool | false : bool.
Definition andb b1 b2 :=
 match b1 with
| true ⇒ b2 | false ⇒ false
 end.
Inductive nat = | O : nat | S : nat → nat.
Fixpoint fact n := match n with
   | O ⇒ S O   | S p → n * fact p
   end.
```

# Typage des types inductifs (première étape)

## l'exemple des booléens

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

*Une telle déclaration de types inductifs définit :*

- un type $\Gamma \vdash \mathit{bool} : \mathbf{Type}$
- un ensemble de règles d'introduction pour ce type : les constructeurs

$$\Gamma \vdash \mathit{true} : \mathit{bool} \qquad \Gamma \vdash \mathit{false} : \mathit{bool}$$

- une règle d'élimination, sous la forme d'un opérateur de filtrage

$$\frac{\Gamma \vdash t : \mathit{bool} \quad \Gamma \vdash A : s \quad \Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : A}{\Gamma \vdash (\texttt{match}\ t\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) : A}$$

- des règles de réduction, dites règles de $\iota$-réduction

$$\begin{array}{lcl} (\texttt{match}\ \mathit{true}\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) & \rightarrow_\iota & t_1 \\ (\texttt{match}\ \mathit{false}\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) & \rightarrow_\iota & t_2 \end{array}$$

# Types inductifs avec paramètres

L'exemple de la disjonction

*en Objective Caml*

```
type ('a,'b) or =
  | or_introl of 'a | or_intror of 'b
```

*dans le CCI, syntaxe Coq*

```
Inductive or (A:Prop) (B:Prop) : Prop :=
  | or_introl : A → or A B
  | or_intror : B → or A B.
```

# Types inductifs avec paramètres

l'exemple de la disjonction

```
Inductive or (A:Prop) (B:Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.
```

*qui définit*

- une famille de type

$$\frac{}{\Gamma \vdash \mathit{or} : \mathbf{Prop} \to \mathbf{Prop} \to \mathbf{Prop}}$$

- un ensemble de règles d'introduction pour les types de cette famille

$$\frac{\Gamma \vdash A : \mathbf{Prop} \quad \Gamma \vdash B : \mathbf{Prop} \quad \Gamma \vdash p : A}{\Gamma \vdash \mathit{or\_introl}_{A,B}\ p : \mathit{or}\ A\ B}$$

$$\frac{\Gamma \vdash A : \mathbf{Prop} \quad \Gamma \vdash B : \mathbf{Prop} \quad \Gamma \vdash q : B}{\Gamma \vdash \mathit{or\_intror}_{A,B}\ q : \mathit{or}\ A\ B}$$

# Disjonction (suite)

une règle d'élimination

$$\frac{\Gamma \vdash t : \mathit{or}\ A\ B \quad \Gamma \vdash C : \mathbf{Prop} \quad \Gamma, p : A \vdash t_1 : C \quad \Gamma, q : B \vdash t_2 : C}{\Gamma \vdash (\texttt{match}\ t\ \texttt{with}\ \mathit{or\_introl}_{A,B}\ p \Rightarrow t_1 \mid \mathit{or\_intror}_{A,B}\ q \Rightarrow t_2\ \texttt{end}) : C}$$

Des règles de $\iota$-réduction

$$(\texttt{match}\ \mathit{or\_introl}_{A,B}\ t\ \texttt{with}\ \mathit{or\_introl}_{A,B}\ p \Rightarrow t_1 \mid \mathit{or\_intror}_{A,B}\ q \Rightarrow t_2\ \texttt{end}) \rightarrow_\iota t_1[t/p]$$

$$(\texttt{match}\ \mathit{or\_intror}_{A,B}\ u\ \texttt{with}\ \mathit{or\_introl}_{A,B}\ p \Rightarrow t_1 \mid \mathit{or\_intror}_{A,B}\ q \Rightarrow t_2\ \texttt{end}) \rightarrow_\iota t_2[u/q]$$

# Remarque sur la syntaxe

En pratique, Coq définit les constructeurs sous leur forme curryfiée (contrairement à Objective Caml où les constructeurs sont nécessairement appliqués).

- Les règles d'introduction pour la disjonction implantées par Coq sont, en fait :

$$\Gamma \vdash \mathit{or\_introl} : \forall A\, B : \mathbf{Prop}, A \to \mathit{or}\ A\ B$$

$$\Gamma \vdash \mathit{or\_intror} : \forall A\, B : \mathbf{Prop}, B \to \mathit{or}\ A\ B$$

- À l'inverse, les paramètres des constructeurs sont omis dans la syntaxe du `match` de Coq (information redondante déjà présente, implicitement, dans le type de l'argument filtré).

$$\frac{\Gamma \vdash t : \mathit{or}\ A\ B \quad \Gamma \vdash C : \mathbf{Prop} \quad \Gamma, p : A \vdash t_1 : C \quad \Gamma, q : B \vdash t_2 : C}{\Gamma \vdash (\texttt{match}\ t\ \texttt{with}\ \mathit{or\_introl}\ p \Rightarrow t_1 \mid \mathit{or\_intror}\ q \Rightarrow t_2\ \texttt{end}) : C}$$

- et les règles de $\iota$-réduction s'écrivent, *en Coq* : :

$$(\texttt{match}\ \mathit{or\_introl}\ A\ B\ t\ \texttt{with}\ \mathit{or\_introl}\ p \Rightarrow t_1 \mid \mathit{or\_intror}\ q \Rightarrow t_2\ \texttt{end}) \to_\iota \quad t_1[t/p]$$

# Types inductifs (élimination dépendante)

l'exemple des booléens

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

- la règle d'élimination dans toute sa généralité

$$\frac{\Gamma \vdash t : \mathit{bool} \quad \Gamma, x : \mathit{bool} \vdash A(x) : s \quad \Gamma \vdash t_1 : A(\mathit{true}) \quad \Gamma \vdash t_2 : A(\mathit{false})}{\Gamma \vdash (\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) : A(t)}$$

- règles de réduction

$$(\texttt{match}\ \mathit{true}\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) \longrightarrow_\iota t_1$$

$$(\texttt{match}\ \mathit{false}\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) \longrightarrow_\iota t_2$$

- On vérifie en particulier la préservation du type par réduction.

# Types inductifs (élimination dépendante)

- Ce schéma permet de dériver l'analyse de cas sur les booléens

$$\lambda P : bool \to \textbf{Prop}.\ \lambda H_{true} : P(true).\ \lambda H_{false} : P(false).\ \lambda x : bool.$$
$$\texttt{match}\ x\ \texttt{as}\ y\ \texttt{with}\ true\ \texttt{=>}\ H_{true} \mid false\ \texttt{=>}\ H_{false}\ \texttt{end}$$

- est une preuve de

$$\forall P : bool \to \textbf{Prop}.\ P(true) \to P(false) \to \forall x : bool.\ P(x)$$

La même en syntaxe Coq :

```
Definition bool_case
  : ∀P : bool → Prop, P true → P false
    → ∀b : bool, P b
  : = fun (P : bool → Prop)
          (Ht : P true) (Hf : P false) (b : bool) ⇒
      match b as b0 return (P b0) with
      | true ⇒ Ht | false ⇒ Hf
      end.
```

# Types inductifs (élimination dépendante)

L'exemple des booléens

- L'élimination dépendante permet aussi de construire des fonctions dans des types produits
- $$\lambda A : bool \to \textbf{Type}.\, \lambda H_{true} : A(true).\, \lambda H_{false} : A(false).\, \lambda x : bool.\, \texttt{match}\ x\ \texttt{as}\ y\ \texttt{return}\ A(y)\ \texttt{with}\ true \Rightarrow H_{true} \mid false \Rightarrow H_{false}\ \texttt{end}$$
- est un combinateur de type :

$$\forall A : bool \to \textbf{Type}.\, A(true) \to A(false) \to \forall x : bool.\, A(x)$$

- Il permet de construire des fonctions dans le type $\Pi x : bool.\, A(x)$.

# Types inductifs avec preuves dépendantes

## L'exemple de la disjonction

```
Inductive or (A:Prop) (B:Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.
```

- la règle d'élimination dans toute sa généralité

$$\frac{\Gamma \vdash t : or\ A\ B \quad \Gamma, x : or\ A\ B \vdash C(x) : \mathbf{Prop} \quad \Gamma, p : A \vdash t_1 : C\,(or\_introl\ p) \quad \Gamma, q : B \vdash t_2 : C\,(or\_intror\ q))}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ C(x)\ \texttt{with} \\ \quad or\_introl\ p \Rightarrow t_1 \mid or\_intror\ q \Rightarrow t_2 \\ \texttt{end}\end{array}\right) : C(t)}$$

# Types inductifs avec preuves dépendantes

L'élimination dépendante :

- permet de raisonner par analyse de cas sur la forme d'une preuve.
  $\lambda P : or\ A\ B \to \mathbf{Type}.$
  $\lambda H_l : (\forall p : A.\ P\ (or\_introl\ p)).$
  $\lambda H_r : (\forall q : B.\ P\ (or\_intror\ q)).\ \lambda x : or\ A\ B.$
  `match` $x$ `as` $y$ `return` $P(y)$ `with`
  $or\_introl\ p \Rightarrow H_l\ p \mid or\_intror\ q \Rightarrow H_r\ q$
  `end`
- est une preuve de :
  $\forall P : (or\ A\ B) \to \mathbf{Prop}.$
  $(\forall p : A,\ P(or\_introl\ p)) \to (\forall q : B,\ P(or\_intror\ q))$
  $\to \forall x : (or\ A\ B).\ P(x)$

# Types inductifs récursifs

l'exemple des entiers

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

*qui définit*

- un type $\Gamma \vdash \texttt{nat} : \textbf{Type}$
- un ensemble de règles d'introduction pour ce type : les constructeurs

$$\Gamma \vdash O : \texttt{nat} \qquad \frac{\Gamma \vdash n : \texttt{nat}}{\Gamma \vdash S\ n : \texttt{nat}}$$

- une règle d'élimination (opérateur de filtrage à résultat dépendant du filtre)

$$\frac{\Gamma \vdash t : \texttt{nat} \quad \Gamma, x{:}\texttt{nat} \vdash A(x) : s \quad \Gamma \vdash t_1 : A(O) \quad \Gamma, n{:}\texttt{nat} \vdash t_2 : A(S\ n)}{\Gamma \vdash (\texttt{match}\ t\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end}) : A(t)}$$

- des règles de réduction préservant le typage ($\iota$-réduction)

$$(\texttt{match}\ O\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end}) \quad \rightarrow_\iota \quad t_1$$
$$(\texttt{match}\ S\ n\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \texttt{end}) \quad \rightarrow_\iota \quad t_2$$

# Types inductifs récursifs

l'exemple des entiers

- On a analyse de cas et construction par cas : le terme

$$
\begin{array}{l}
\lambda P : \texttt{nat} \to s. \\
\lambda H_O : P(O). \\
\lambda H_S : \forall m : \texttt{nat}. P(S\ m). \\
\lambda n : \texttt{nat}. \\
\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with} \\
\quad\ |\ O\ \texttt{=>}\ H_O \\
\quad\ |\ S\ m\ \texttt{=>}\ H_S\ m \\
\quad \texttt{end}
\end{array}
$$

- est une preuve de

$$\forall P : \texttt{nat} \to s. P(O) \to (\forall m : \texttt{nat}. P(S\ m)) \to \forall n : \texttt{nat}. P(n)$$

# Types inductifs avec paramètres

l'exemple des listes

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

*qui définit*

- une famille de type $\dfrac{}{\Gamma \vdash \mathit{list} : \mathbf{Type} \to \mathbf{Type}}$
- un ensemble de règles d'introduction pour les types de cette famille

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathtt{nil}_A : \mathit{list}\ A} \qquad \frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash l : \mathit{listA}}{\Gamma \vdash \mathtt{cons}_A\ a\ l : \mathit{list}\ A}$$

# Types inductifs avec paramètres

## l'exemple des listes : élimination

- une règle d'élimination (opérateur de filtrage à résultat dépendant du filtre)

$$\frac{\Gamma \vdash l : \mathit{list}\ A \quad \Gamma, x : \mathit{list}\ A \vdash C(x) : s \qquad \Gamma \vdash t_1 : C(\mathtt{nil}) \quad \Gamma, a : A, l : \mathit{list}\ A \vdash t_2 : C(\mathtt{cons}_A\ a\ l)}{\Gamma \vdash \left(\begin{array}{l}\mathtt{match}\ l\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil} \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right) : C(l)}$$

- des règles de réduction qui préservent le typage ($\iota$-réduction)

$$\begin{array}{l}\left(\begin{array}{l}\mathtt{match}\ \mathtt{nil}_A\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil} \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right)\\ \rightarrow_\iota \quad t_1\end{array}$$

$$\begin{array}{l}\left(\begin{array}{l}\mathtt{match}\ \mathtt{cons}_A\ a'\ l'\ \mathtt{as}\ x\ \mathtt{return}\ C(x)\ \mathtt{with}\\ \quad \mathtt{nil}\ p \Rightarrow t_1 \mid \mathtt{cons}\ a\ l \Rightarrow t_2\\ \mathtt{end}\end{array}\right)\\ \rightarrow_\iota \quad t_2[a'/a; l'/l]\end{array}$$

# Types inductifs avec paramètres et index

*l'exemple des vecteurs avec leur taille*

```
Inductive vect (A:Type) : nat → Type :=
| niln : vect A O
| consn : A → ∀n:nat, vect A n → vect A (S n).
```

*qui définit*

- une famille de types-prédicats : $\Gamma \vdash \mathit{vect} : \mathbf{Type} \to \mathit{nat} \to \mathbf{Type}$
- un ensemble de règles d'introduction pour les types de cette famille

$$\frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash \mathtt{niln}_A : \mathit{vect}\ A\ O}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash n : \mathit{nat} \quad \Gamma \vdash l : \mathit{vect}\ A\ n}{\Gamma \vdash \mathtt{consn}_A\ a\ n\ l : \mathit{list}\ A\ (S\ n)}$$

# Types inductifs avec paramètres et index

*vecteurs : élimination*

- une règle d'élimination (opérateur de filtrage à résultat dépendant du filtre)

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathit{vect}\ A\ n \qquad \Gamma, m{:}\mathit{nat}, x{:}\mathit{vect}\ A\ m \vdash C(m,x) : s \\ \Gamma \vdash t_1 : C(O, \mathtt{niln}_A) \\ \Gamma, a{:}A, n{:}\mathit{nat}, l{:}\mathit{vect}\ A\ n \vdash t_2 : C(S\ n, \mathtt{consn}_A\ a\ n\ l)\end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathtt{match}\ v\ \mathtt{as}\ x\ \mathtt{in}\ \mathit{vect}\ \_\ p\ \mathtt{return}\ C(p,x)\ \mathtt{with} \\ \quad \mathtt{niln} \Rightarrow t_1 \mid \mathtt{consn}\ a\ n\ l \Rightarrow t_2 \\ \mathtt{end}\end{array}\right) : C(n,v)}$$

- des règles de réduction qui préservent le typage ($\iota$-réduction)

$$\begin{array}{l}\left(\begin{array}{l}\mathtt{match}\ \mathtt{niln}_A\ \mathtt{as}\ x\ \mathtt{in}\ \mathit{vect}\ \_\ p\ \mathtt{return}\ C(x,p)\ \mathtt{with} \\ \quad \mathtt{niln} \Rightarrow t_1 \mid \mathtt{consn}\ a\ n\ l \Rightarrow t_2 \\ \mathtt{end}\end{array}\right) \\ \quad \rightarrow_\iota \quad t_1 \\ \left(\begin{array}{l}\mathtt{match}\ \mathtt{consn}_A\ a'\ n'\ l'\ \mathtt{as}\ x\ \mathtt{in}\ \mathit{vect}\ \_\ p\ \mathtt{return}\ C(x,p)\ \mathtt{with} \\ \quad \mathtt{niln} \Rightarrow t_1 \mid \mathtt{consn}\ a\ n\ l \Rightarrow t_2 \\ \mathtt{end}\end{array}\right) \\ \quad \rightarrow_\iota \quad t_2[a'/a; n'/n; l'/l]\end{array}$$