Proof assistants

Bruno Barras, Guillaume Melquiond and Christine Paulin-Mohring

Université Paris-Sud 11, INRIA

07/12/10

Objectives

- Study proof assistants (interactive construction of proofs) based on higher-order type theory and more specifically the system Coq.
  - How to build/how to use an environment for developing formal proofs on computer.
- Study inductive definitions
  - Theory and practice
- Application to proof of programs.
  - Functional programming with dependent types
  - Modeling imperative programs

Practical informations

- WEB page for the course (course notes, slides, exercises with solutions, old projects and exams) :
  `http://www.lri.fr/~paulin/MPRI`
- Bruno Barras, projet Typical, INRIA Saclay - Île-de-France and LIX, Ecole Polytechnique
  Bruno.Barras@inria.fr
- Guillaume Melquiond, projet PROVAL, INRIA Saclay - Île-de-France and LRI, Université Paris-Sud
  Guillaume.Melquiond@lri.fr
- Christine Paulin, projet PROVAL, LRI, Université Paris-Sud and INRIA Saclay - Île-de-France
  Christine.Paulin@lri.fr

Organisation

Two hours lecture + 1 hour Coq practice on computers (room 1C22)

Evaluation

- Classical written exam.
- An optional project
  may count for half of the final grade $max(E, \frac{E+P}{2})$
  A good training for the exam
- The projet is done with Coq
  Expected result : source code, small report and an individual defense (10-15mn).
  Subject given after christmas

Plan

- 07/12 - CP - Introduction to Coq theory, Inductive Definitions 1
- 14/12 - CP - Inductive Definitions 2
- 04/01 - BB - Functional Programming 1, structural versus well-founded induction, partial function, coinductive definitions.
- 11/01 - BB - Functional Programming 2, monadic constructions, modules. Models, realisability, extraction.
- 18/01 - BB - Architecture of a proof assistant, automated versus interactive proofs, tactic language
- 25/01 - GM - Proof of imperative programs
- 01/02 - GM - Automated proofs. Floating point arithmetic.
- 08/02 - support for project
- 15/02 - GM - Proof by reflexion (example on intervals).
- 01/03 or 08/03 - Exam + project defense

Plan

Introduction to the Calculus of Inductive Constructions

- Proof Assistants
- From the Calculus of Constructions to the Calculus of Inductive Constructions
- Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions

- Fixpoint operators
- Conditions for inductive definitions
- Advanced inductive definitions

Summary

Introduction to the Calculus of Inductive Constructions
Proof Assistants
From the Calculus of Constructions to the Calculus of Inductive Constructions
Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
Fixpoint operators
Conditions for inductive definitions
Advanced inductive definitions

Proofs on computers

For doing proofs with computers we need :

- A language to represent objects : integers, functions, sets, ...
- A language to represent properties of objects : first-order logic, higher-order logic.
- A method to construct/verify proofs (basic rules + a way to mechanize them).

Approach based on higher-order logic :

- typed lambda-calculus for representing objects and properties $\neq$ set theory (first order)
- tactics or well-typed proof terms for builing and verifying proofs.

Examples of case studies

In the Coq proof assistant but analogous examples in Isabelle/HOL

- Formalisation of semantics of languages such as JavaCard, certification of security functionalities (Gemplus, Trusted Logic)
- Proof of the 4-colors theorem (G. Gonthier, B. Werner - INRIA - Microsoft Research)
- Development of a certified C compiler producing optimized code (Compcert, X. Leroy)
- Formalisation and reasoning on floating-point number arithmetic (S. Boldo, G. Melquiond . . .)
- Development of certified static analysers (D. Pichardie)
- . . .

Summary

Introduction to the Calculus of Inductive Constructions
Proof Assistants
From the Calculus of Constructions to the Calculus of Inductive Constructions
Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
Fixpoint operators
Conditions for inductive definitions
Advanced inductive definitions

History (1)

- Calculus of Constructions (Coquand-Huet, 1984)
  - Abstraction/Application/Product as only operators (PTS)
  - A unique sort **Prop** for representing types and propositions
  - All products where possible : polymorphism $\forall A : \mathbf{Prop}, A \to A$, dependent types $P : A \to \mathbf{Prop}$, $P : \mathbf{Prop} \to \mathbf{Prop}$
  - Representing data and properties using impredicative encodings (Church's integers, Leibniz equality).
- A hierarchy of universes is added (Coquand, 1986). More polymorphism : $A : \mathbf{Type}$ can be instantiated by **Prop**, $A \to \mathbf{Prop}$, $\mathbf{Prop} \to \mathbf{Prop} \ldots$)
- A distinction **Prop**, **Set** is added between logical properties and computational properties (program extraction, 1989).

History (2)

Inductive Definitions :

- Martin-Löf Type Theory (1984) : no impredicativity but basic inductive constructions added following a general scheme : rules for construction, elimination, computation.
- Calculus of Inductive Constructions (Coquand-Paulin, 1991). A tentative to merge the two formalisms :
  - (co)-inductives primitive definitions
    easy to use (less encoding than with impredicativity) and their generality (computational and logical properties)
  - An higher-order logic for more expressivity.

The structure of the Calculus of Inductive Constructions

Calculus of Inductive Constructions (predicative – Coq $\geq$ 8.0)

=

Calculus of Constructions on **Prop** and **Type**

for higher-order logic

for impredicative types (historically)

+

Type hierarchy **Set** : **Type** = $\mathbf{Type}_1$: $\mathbf{Type}_2$: $\mathbf{Type}_3$ ...

for program extraction

for logical expressivity

+

Inductive types

for more « natural » formalisations and data-types

more computational expressivity

more logical expressivity

Reminder on Pure Type Systems (PTS)

- Atoms : sorts (types of types), organised in axioms $\mathcal{A}$ and rules for product $\mathcal{R}$, Variables ;
- product types $\Pi x : A.B$ (or $\forall x : A.B$) with A and B types ; written $A \to B$ when x is not free in B ;
- Abstraction $\lambda x : A.t$; Application $t\, u$

Rules

$$\frac{\Gamma\ \mathbf{ok} \quad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A\ \mathbf{ok}} \qquad \frac{\Gamma\ \mathbf{ok} \quad (x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : s_3}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \qquad \frac{\Gamma \vdash t : \Pi x : A.B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[x \leftarrow u]}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A \equiv B}{\Gamma \vdash t : B} \qquad \lambda x : A.t\, u \equiv t[x \leftarrow u]$$

System F seen as (second-order) propositionnal logic

Axiom $\mathcal{A} = \{\mathbf{Prop} : \mathbf{Type}\}$,
Rules $\mathcal{R} = \{(\mathbf{Prop}, \mathbf{Prop}, \mathbf{Prop}); (\mathbf{Type}, \mathbf{Prop}, \mathbf{Prop})\}$

System F "propositional"

$\Pi A : \mathbf{Prop}.\, B$	$\forall A : \mathbf{Prop}, B$
$A \to B$	$A \Rightarrow B$
$\Pi C : \mathbf{Prop}.\, (A \to B \to C) \to C$	conjunction $A \wedge B$
$\Pi C : \mathbf{Prop}.\, (\forall A : \mathbf{Prop}, B \to C) \to C$	existential $\exists A : \mathbf{Prop}, B$

and abstraction, application, and variables
implement inference rules for the logic

System F as a calculus

Polymorphic Lambda-calculus (second order)

System F "computational"

$\Pi A : \textbf{Prop}.\ B$	$\forall A : \textbf{Set},\ B$
$A \to B$	$A \to B$
$\Pi C : \textbf{Prop}.\ (A \to B \to C) \to C$	product $A \times B$
$\lambda A : \textbf{Prop}.\ t$	**fun**(A :**Set**) ⇒t
$\lambda x : A.\ t$	**fun**(x :A) ⇒t
$t\ A$	$t\ A$
$t\ u$	$t\ u$
x	x
$\lambda C : \textbf{Prop}.\lambda f : A \to B \to C.f\ a\ b$	pair (a, b)

From System F to the Calculus of Constructions

- Goal : be able to talk about the computational part of System F inside the logical part of the system.
- Add product of the form $(\textbf{Prop}, \textbf{Type}, \textbf{Type})$

$$A \to \textbf{Prop} \qquad\qquad P : A \to \textbf{Prop},\ P\,t : \textbf{Prop}$$

- ... and add higher-order polymorphism ; products with the form $(\textbf{Type}, \textbf{Type}, \textbf{Type})$

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

From System F to the Calculus of Constructions (2)

- $\hookrightarrow$ The Calculus of Constructions implements the Curry-Howard-de Bruijn correspondance as an identity.
- $\hookrightarrow$ An original logic which can “speak of” proofs.
- It is possible to “forget” proof terms : rule $(\mathbf{Prop}, \mathbf{Type}, \mathbf{Type})$
  Consequence : conservativity of CC over F_ω.
  With $A : \mathbf{Prop}$, $K : \mathbf{Type}$, $P : K : \mathbf{Type}$ and $t : A : \mathbf{Prop}$.

$$\Pi x : A.K \longrightarrow K \qquad P\,t \longrightarrow P \qquad \lambda x : A.P \longrightarrow P$$

The Calculus of Constructions : a complex system

Computational and logic levels are superposed

$\hookrightarrow$ introduction of **Set** next to **Prop**
duplication of System F level with the following intended meaning :

- in **Prop**, the form of proofs does not matter ; the principle of indiscernability ($\forall P : \mathbf{Prop}. \forall pq : P.\, p = q$) is admissible.
- in **Set**, the objets can be discrimined ; for instance in the type of booleans, $\texttt{true} \neq \texttt{false}$ will be admissible.
- **Prop** can be interpreted as a boolean type : a proposition which is provable is interpreted by `true` and a proposition which is provably false is interpreted by `false`.
- We can encode the natural numbers but we cannot prove $0 \neq 1$ (because we can forget about type depending on terms)

$$(\forall P. P\, 0 \to P\, 1) \to \forall C. C$$

System U

what if the type level of System F is polymorphic and impredicative

Adding variables of type **Type** .

- Adding the axiom : $(\textbf{Type}, \textbf{Type}')$

$$\frac{K : \textbf{Type} \vdash A : \textbf{Prop}}{\Pi K : \textbf{Type}.\, A : \textbf{Prop}} \quad (\textbf{Type}', \textbf{Prop}, \textbf{Prop})$$

$$\Pi K : \textbf{Type}.\, K : \textbf{Type} \qquad (\textbf{Type}', \textbf{Type}, \textbf{Type})$$

- ... we obtain a provably inconsistent system :
  - encoding of Burali-Forti paradox (Girard 1978),
  - Russell paradox (Miquel 2000),
  - even a quasi fixpoint (Hurkens).
- reasoning on proofs of an impredicative system of predicates ... is inconsistent

System $F_{\omega.2}$

what if the type level of F_ω is simply polymorphic but predicative

- Adding $\mathbf{Type}_2$ on top of **Type** introduces polymorphism at the type level of system F_ω but without impredicativity

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

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

- ... logical strength is equivalent to Zermelo set theory
- In particular : we can define integers with $0 \neq 1$ provable.
- Natural generalisation : a hierarchy of universes

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

- ... adding types depending on proofs, we obtain the calculus of constructions extended with universes.

Drawbacks of polymorphic encoding of inductive definitions

Case of impredicative encoding

- $0 \neq 1$ is not provable
- induction is not ≪ directly ≫ provable (only the recursor is available)
- *Case of predicative encoding in the calculus with universes*
  - OK for expressivity (we have $0 \neq 1$ and an ≪ indirect ≫ induction)
  - *But* no predecessor in 1 step
  - not “natural”
  - difficult to write automated tools that can distinguish between inductive types constructors and arbitrary terms
- Primitive inductive types ≪ à la Martin-Löf ≫ have been added.

The Calculus of Inductive Constructions (Coq $\geq$ 5.6)

A general scheme for building inductive types

- positivity criteria (to ensure the existence of a smallest subset which contains a given set of constructors)
- recursors (like in Gödel system T) are decomposed into an operator for pattern-matching (`match-with`) and a fixpoint combinator (`fix`)
- syntactic criteria for terminaison of fix-points
- *Specific elimination conditions according to sorts*
  - respect computational interpretation of **Set** and **Type** and the purely logical interpretation of **Prop**
  - avoid paradoxes related to impredicativity
- *A few consequences*
  - $0 \neq 1$ is derivable
  - induction principle is derivable
  - intuitionistic choice axiom is derivable

The limits of the Calculus of Inductive Constructions

- **Set** impredicativity at the computational level gives to the Calculus of Inductive Constructions (CCI) a strong intuitionnistic flavor (only computational models)
- Choice axiom with classical logic are inconsistent, extensionnality of functions is not validated
- Limits the possibility to formalise classical mathematics
- Choice : change Coq default behavior : CCI with **Set** predicative Rule $(\mathbf{Type}, \mathbf{Set}, \mathbf{Type})$:$\Pi X : \mathbf{Set}.X : \mathbf{Type}$.

Calculus of Predicative Inductive Constructions

Coq $\geq$ 8.0

- Sort **Set** added to the hierarchy of types (**Set** = **Type**$_0$)
- no difference (except for historical reasons) between data-types in **Set** or in **Type**.
- An approach closer to the HOL system (but with inductive types and a hierarchy of universes)
- Compatible with the standard mathematical axioms : classical logic, classical choice axiom, extensionnality (justified by embedding into set theory)

Summary

Introduction to the Calculus of Inductive Constructions
- Proof Assistants
- From the Calculus of Constructions to the Calculus of Inductive Constructions
- Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
- Fixpoint operators
- Conditions for inductive definitions
- Advanced inductive definitions

Inductive types : booleans

in Objective Caml

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

System F part of 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.
```

as an inductive primitive type in the Calculus of Inductive Constructions

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

inductive types : booleans

in Objective Caml

```
type bool = | true | false
let orb b1 b2 = match b1 with
   | true -> true | false -> b1
```

in System F, Coq syntax

```
Definition bool := ∀P:Prop, P → P → P.
Definition true : bool := fun P:Prop ⇒ fun H1 H2 ⇒ H1.
Definition orb (b1 b2 :bool) : bool
    := b1 bool true b2.
```

in CCI, Coq syntax

```
Inductive bool : Type := | true : bool | false : bool.
Definition orb b1 b2 :=
   match b1 with
  | true ⇒ true | false ⇒ b2
   end.
```

Inductive types : natural numbers

in Objective Caml

```
type nat = | O | S of nat
let rec fact n =  match n with
    | O -> S(O)  | S(p) -> n * fact p
```

in CCI, Coq syntax

```
Inductive nat : Type := | O : nat | S : nat → nat.
Fixpoint fact n := match n with
  | O ⇒ S O  | S p → n * fact p
  end.
```

Typing inductive types (first step)

Booleans example

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

Such a declaration defines :

- a type $\Gamma \vdash \mathit{bool} : \mathbf{Type}$
- a set of introduction rules for this type : constructors

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

- an elimination rule, as a pattern-matching operator

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

- reduction rules, (a.k.a. ι-reduction)

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

Inductive types with parameters

Example of disjonction

in Objective Caml

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

in CCI, Coq syntax

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

Inductive types with parameters

Example of disjunction

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

which defines

- a family of types

$$\frac{}{\Gamma \vdash or : \textbf{Prop} \rightarrow \textbf{Prop} \rightarrow \textbf{Prop}}$$

- a set of introduction rules for the types in this family

$$\frac{\Gamma \vdash A : \textbf{Prop} \quad \Gamma \vdash B : \textbf{Prop} \quad \Gamma \vdash p : A}{\Gamma \vdash or\_introl_{A,B}\ p : or\ A\ B}$$

$$\frac{\Gamma \vdash A : \textbf{Prop} \quad \Gamma \vdash B : \textbf{Prop} \quad \Gamma \vdash q : B}{\Gamma \vdash or\_intror_{A,B}\ q : or\ A\ B}$$

Disjunction (2)

an elimination rule

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

Rules for ι-reduction

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

Remark on the syntax

Coq defines constructors in a curryfied way (in Objective Caml, a constructor is allways applied to arguments)

- introduction rules for disjunction implanted by Coq are :

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

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

- On the opposite the constructors parameters are ommited in the syntax of patterns in a `match` (information found in the type of the filtered argument).

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

- the rules of ι-reduction can be written, *in Coq* : :

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

Inductive types (dependent elimination)

Booleans example

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

- The general elimination rule :

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

- Reduction rule

$$\begin{array}{l}(\texttt{match}\ \mathit{true}\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) \\ \rightarrow_\iota \quad t_1 \\ (\texttt{match}\ \mathit{false}\ \texttt{as}\ x\ \texttt{return}\ A(x)\ \texttt{with}\ \mathit{true} \Rightarrow t_1 \mid \mathit{false} \Rightarrow t_2\ \texttt{end}) \\ \rightarrow_\iota \quad t_2\end{array}$$

- We check in particular that types are preserved by reduction.

Inductive types (dependent elimination)

- From this scheme we get case analysis on booleans

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

- is a proof of

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

Same using Coq syntax :

```
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.
```

Inductive types (dependent elimination

Boolean example

- Dependent elimination also gives the possibility to construct functions in product types
- $$\lambda A : bool \to \textbf{Type}.\, \lambda H_{true} : A(true).\, \lambda H_{false} : A(false).\, \lambda x : bool.$$
  $$\texttt{match}\ x\ \texttt{as}\ y\ \texttt{return}\ A(y)\ \texttt{with}\ true \Rightarrow H_{true} \mid false \Rightarrow H_{false}\ \texttt{end}$$
- is a combinator of type :

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

- It allows to build functions in the type $\Pi x : bool.\, A(x)$.

```
Definition A x :=
   match x with true ⇒ nat | false ⇒ bool end.
Definition F x : A x :=
   match x return A x with
        true ⇒ O | false ⇒ false
   end.
```

Inductive types with dependent proofs

Disjunction example

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

- General elimination rule

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

Inductive types with dependent proofs

Dependent elimination :

- allows to reason by case on the form of a proof.

  $\lambda P : \mathit{or}\ A\ B \to \textbf{Prop}.$
  $\lambda H_l : (\forall p : A.\, P\,(\mathit{or\_introl}\ p)).$
  $\lambda H_r : (\forall q : B.\, P\,(\mathit{or\_intror}\ q)).\, \lambda x : \mathit{or}\ A\ B.$
  $\quad \texttt{match}\ x\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with}$
  $\qquad \mathit{or\_introl}\ p \Rightarrow H_l\ p \mid \mathit{or\_intror}\ q \Rightarrow H_r\ q$
  $\quad \texttt{end}$

- is a proof of :

  $\forall P : (\mathit{or}\ A\ B) \to \textbf{Prop}.$
  $(\forall p : A,\ P(\mathit{or\_introl}\ p)) \to (\forall q : B,\ P(\mathit{or\_intror}\ q))$
  $\to \forall x : (\mathit{or}\ A\ B).\, P(x)$

Recursive inductive types

Natural numbers example

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

which defines

- a type $\Gamma \vdash \texttt{nat} : \textbf{Type}$
- a set of introduction rules for this type : constructors

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

Recursive inductive types : Natural numbers example

which defines also

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

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

- reduction rules preserve typing (ι-reduction)

$$(\mathtt{match}\ O\ \mathtt{as}\ x\ \mathtt{return}\ A(x)\ \mathtt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \mathtt{end}) \rightarrow_\iota t_1$$
$$(\mathtt{match}\ S\ m\ \mathtt{as}\ x\ \mathtt{return}\ A(x)\ \mathtt{with}\ O \Rightarrow t_1 \mid S\ n \Rightarrow t_2\ \mathtt{end}) \rightarrow_\iota t_2[m/n]$$

Recursive inductive types

Example of natural numbers

- We obtain case analysis and construction by cases : the term

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

- is a proof of

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

Inductive types with parameters

Example of lists

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

which defines

- a family of types $\dfrac{}{\Gamma \vdash \mathit{list} : \mathbf{Type} \to \mathbf{Type}}$
- a set of introduction rules for the types in this family

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

Inductive types with parameters

Example of lists : elimination

- An elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

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

- reduction rules which preserves typing (ι-reduction)

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

Inductive types with parameters and index

Example of vectors with size

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

which defines

- a family of types-predicates : $\Gamma \vdash \mathit{vect} : \mathbf{Type} \rightarrow \mathit{nat} \rightarrow \mathbf{Type}$
- a set of introduction rules for the types in this family

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

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

Inductive types with parameters and index

vectors : elimination

- an elimination rule (pattern-matching operator with a result depending on the object which is eliminated)

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

- reduction rules preserve typing (ι-reduction)

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

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

Inductive Definitions II - Dec 14th 2010

Introduction to the Calculus of Inductive Constructions
Proof Assistants
From the Calculus of Constructions to the Calculus of Inductive Constructions
Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
Fixpoint operators
Conditions for inductive definitions
Advanced inductive definitions

Summary

Introduction to the Calculus of Inductive Constructions
- Proof Assistants
- From the Calculus of Constructions to the Calculus of Inductive Constructions
- Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
- Fixpoint operators
- Conditions for inductive definitions
- Advanced inductive definitions

Recursive inductive types : example of natural numbers

Case analysis and construction by case : the term

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

is a proof of

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

How to derive the standard recursion scheme ?

Fixpoint operator (first step)

We add an anonymous typed fixpoint construction

$$(\texttt{fix}\, f\ (x:A) : B := t(f,x))$$

...the type of the result may depend on the argument

$$(\texttt{fix}\, f\ (x:A) : B(x) := t(f,x))$$

Comparison with `let rec` à la ML (named fixpoint)

$$\begin{array}{c}(\texttt{fix}\, f\ (x:A) : B(x) := t(f,x)) \\ = \\ \texttt{let rec}\ f\ (x:A) = t(f,x)\ \texttt{in}\ f\end{array}$$

Coq has a specific construction for named fixpoints :

```
Fixpoint f (x:A) := t.
```

The fixpoint operator (reduction)

Fixpoint expression with dependent result

$$(\mathtt{fix}\, f\ (x : A) : B(x) := t(f, x))$$

- Typing

$$\frac{f : (\forall (x : A), B(x)), x : A \vdash t : B(x)}{\vdash (\mathtt{fix}\, f\ (x : A) : B(x) := t(f, x)) : \forall (x : A), B(x)}$$

- Reduction rule (first approximation) : unfold the fixpoint

$$\begin{gathered}(\mathtt{fix}\, f\ (x : A) : B(x) := t)\ u \\ \longrightarrow \\ t[\mathtt{fix}\, f\ (x : A) : B(x) := t, u/f, x]\end{gathered}$$

Fixpoint operator : application

From case analysis to recursor on natural numbers

case-analysis

$\lambda P : \texttt{nat} \to s,$
$\lambda H_O : P(O),$
$\lambda H_S : \forall m : \texttt{nat}, P(S\ m),$
$\lambda n : \texttt{nat},$
$\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with}$
$\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m$
$\quad \texttt{end}$

has type

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

recursor

$\lambda P : \texttt{nat} \to s,$
$\lambda H_O : P(O),$
$\lambda H_S : \forall m : \texttt{nat}, P(m) \to P(S\ m),$
$\texttt{fix}\ f\ (n : \texttt{nat})\ :\ P(n)\ :=$
$\quad \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ P(y)\ \texttt{with}$
$\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\ (f\ m)$
$\quad \texttt{end}$

has type

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

Fixpoint operator : the termination problem

Implementation in the Calculus of Inductive Constructions :

- built on decidability of typing and conversion
- must forbid unfolding fixpoints ad infinitum

Consistency of the Calculus of Inductive Constructions :

- must forbid infinite proofs such that
  $(\texttt{fix}\ f\ (n : nat) : False := f\ n) : False$

$\hookrightarrow$ choice to require a syntactic criteria for well-founded fixpoints.

Fixpoint operator : well-foundness

Requirement of the Calculus of Inductive Constructions :

- the argument of the fixpoint has type an inductive definition
- recursive calls are on arguments which are *structurally* smaller

Example of recursor on natural numbers

$$
\begin{array}{l}
\lambda P : \mathtt{nat} \to s, \\
\lambda H_O : P(O), \\
\lambda H_S : \forall m : \mathtt{nat}, P(m) \to P(S\ m), \\
\mathtt{fix}\ f\ (n : \mathtt{nat})\ :\ P(n)\ := \\
\quad \mathtt{match}\ n\ \mathtt{as}\ y\ \mathtt{return}\ P(y)\ \mathtt{with} \\
\qquad O \Rightarrow H_O \mid S\ m \Rightarrow H_S\ m\ (f\ m) \\
\quad \mathtt{end}
\end{array}
$$

is correct with respect to CCI : recursive call on m which is structurally smaller than n in the inductive `nat`.

Fixpoint operator : typing rules

$$\frac{I \text{ inductif} \quad \Gamma \vdash I : s \quad \Gamma, x : A \vdash C : s \quad \Gamma, x : I, f : (\forall x : I, C) \vdash t : C \quad t|_f^{\emptyset} <_I x}{\Gamma \vdash (\texttt{fix}\ f\ (x : I) : C := t) : \forall x : I, C}$$

the main definition of $t|_f^{\rho} <_I x$ are :

$$\frac{z \in \rho \cup \{x\} \quad (u_i|_f^{\rho} <_I x)_{i=1\dots n} \quad A|_f^{\rho} <_I x \quad (t_i|_f^{\rho \cup \{x \in \vec{x_i} | x : \forall y : \vec{U}.I\,\vec{u}\}} <_I x)_i}{\texttt{match}\ z\ u_1 \dots u_n\ \texttt{return}\ A\ \texttt{with} \dots\ c_i\ \vec{x_i} \Rightarrow t_i\ \dots \texttt{end}|_f^{\rho} <_I x}$$

$$\frac{t \neq (z\ \vec{u}) \text{ pour } z \in \rho \cup \{x\} \quad t|_f^{\rho} <_I x \quad A|_f^{\rho} <_I x \quad \dots\ t_i|_f^{\rho} <_I x\ \dots}{\texttt{match}\ t\ \texttt{return}\ A\ \texttt{with} \dots\ c_i\ \vec{x_i} \Rightarrow t_i\ \dots \texttt{end}|_f^{\rho} <_I x}$$

$$\frac{y \in \rho}{f\ y|_f^{\rho} <_I x} \qquad \frac{f \notin t}{t|_f^{\rho} <_I x}$$

+ contextual rules ...

Remarks on the criteria

- Cover simply the schema of primitive recursive definitions and proofs by induction
  Recursive call on all immediate subterms :

$$
\begin{array}{l}
\lambda P : \texttt{list}\, A \to s, \\
\lambda f_1 : P\, \texttt{nil}, \\
\lambda f_2 : \forall (a : A)(l : \texttt{list}\, A), P\, l \to P\, (\texttt{cons}\, a\, l), \\
\texttt{fix}\ \mathit{Rec}\ (x : \texttt{list}\, A)\ :\ P\, x\ := \\
\quad \texttt{match}\ x\ \texttt{return}\ P\, x\ \texttt{with} \\
\qquad \texttt{nil} \Rightarrow f_1 \mid (\texttt{cons}\, a\, l) \Rightarrow f_2\, a\, l\, (\mathit{Rec}\, l) \\
\quad \texttt{end}
\end{array}
$$

- has type

$$
\begin{array}{l}
\forall P : \texttt{list}\, A \to s, \\
P\, \texttt{nil}, \to \\
(\forall (a : A)(l : \texttt{list}\, A), P\, l \to P\, (\texttt{cons}\, a\, l)) \to \\
\forall (x : \texttt{list}\, A), P\, x
\end{array}
$$

Remarks on the criteria

Possibility of recursive call on deep subterms

```
Fixpoint mod2 (n:nat) : nat :=
   match n with O ⇒ O | S O ⇒ S O
              | S (S x) ⇒ mod2 x
   end
```

Possibility of recursive call on terms build by case analysis if each branch is a strict subterm :

```
Definition pred (n:nat) : n<>0→nat:=
 match n return n<>0→nat with
     S p ⇒ (fun (h:S p<>0) ⇒ p)
   | O   ⇒ (fun (h:0<>0) ⇒
             match h (refl_equal 0) return nat with end
            )
 end
Fixpoint F (n:nat) : C :=
  match iszero n with
    (left (H:n=O)) ⇒ ...
  | (right (H:n<>0)) ⇒ F (pred n H)
  end
```

Remarks on the criteria

Note : only the recursive arguments with the *same* type are considered recursive (otherwise paradox related to impredicativity)

```
Inductive Singl (A:Prop) : Prop := c : A → Singl A.
Definition T : Prop := ∀(A:Prop), A → A.
Definition t : T := fun A x ⇒ x.
Fixpoint f (x : Singl T) : bool :=
     match x with (c a) ⇒ f (a (Singl T) (c T t)) end.
```

$$f\,(c\;T\;t) \longrightarrow f\,(t\,(\mathit{Singl}\;T)\,(c\;T\;t)) \longrightarrow f\,(c\;T\;t)$$

Summary

Introduction to the Calculus of Inductive Constructions

Proof Assistants

From the Calculus of Constructions to the Calculus of Inductive Constructions

Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions

Fixpoint operators

Conditions for inductive definitions

Advanced inductive definitions

Terminology

- The Calculus of predicative Inductive Constructions has sorts **Prop**, **Set** $=$ **Type**$_0$, **Type**$_1$, **Type**$_2$, $\ldots$
- **Prop** and **Set** are said small (because they do not type another sort)
- sorts **Type**$_i$ (for $i \geq 1$) are said large (because they type **Prop** and **Set**)

Inductive definitions : positivity condition

Condition of strict positivity. The recursive argument of a constructor of the inductive definition I has type

$$\forall(z_1 : C_1)\ldots(z_k : C_k).I\, t_1 \ldots t_n$$

Example of a non monotonic inductive definition which contradicts normalisation :

```
Inductive lambda : Type :=
| Lam : (lambda → lambda) → lambda
```

We define :

```
Definition app (x y:lambda)
   := match x with (Lam f) ⇒ f y end.
Definition Delta := Lam (fun x ⇒ app x x).
Definition Omega := app Delta Delta.
```

and the evaluation of Ω loops.

Inductive definitions : positivity condition

- An inductive type is defined as the smallest type generated by a set of constructors.
- We can see it as $\mu_X, \oplus_{1 \leq i \leq n} \Gamma_i(X)$ (with μ a fixpoint operator on types) and the existence of this smallest type can be proved at the impredicative level when the operator $\lambda X, \oplus_{1 \leq i \leq n} \Gamma_i(X)$ is monotonic.
- It is sufficent for X to appear only in positive position.
- In pratice, we require strict positivity (X never appears on the left of an arrow, even in a positivity position).
  Strict positivity avoids the encoding of Russell paradox (in **Type**) and is often sufficent for applications.

Inductive : strict positivity condition

Monotonicity is sufficent at the impredicative level :

$$\mu F := \forall (X : \textbf{Prop}), (F\,X \to X) \to X$$

But problematique at level **Type**.

```
Inductive X : Type := inj : ((X → Prop) → Prop) → X.
```

$$\begin{array}{lcl}
P_0 & \triangleq & \lambda x : X, \exists P', x = in(\lambda (P : X \to \textbf{Prop}), P = P') \wedge \neg P'(x) \\
x_0 & \triangleq & inj(\lambda (P : X \to \textbf{Prop}), P = P_0)
\end{array}$$

$$\begin{array}{lcl}
P_0(x_0) & \leftrightarrow & \exists P', x_0 = inj(\lambda P.P = P') \wedge \neg P'(x_0) \\
 & \leftrightarrow & \exists P', inj(\lambda P, P = P_0) = inj(\lambda P.P = P') \wedge \neg P'(x_0) \\
 & \leftrightarrow & \exists P', P' = P_0 \wedge \neg P'(x_0) \\
 & \leftrightarrow & \exists P', P' = P_0 \wedge \neg P_0(x_0) \\
 & \leftrightarrow & \neg P_0(x_0)
\end{array}$$

Conditions on sorts for the inductive definitions

- arity and sort of the inductive definition $I : \forall(x_1 : A_1) \ldots (x_n : A_n) s$
- a constructor has the form $c : \forall(y_1 : B_1) \ldots (y_p : B_p) I\, u_1 \ldots u_n$
- typing condition

$$I : (x_1 : A_1) \ldots (x_n : A_n) s \vdash \forall(y_1 : B_1) \ldots (y_p : B_p) I\, u_1 \ldots u_n : s$$

- The sort of a predicative inductive definition (in the hierarchy **Type**) is the maximum of sorts of the types of the arguments of these constructors.
- The sort of a impredicative inductive definition (type **Prop**) has no constraint.

```
Inductive PB : Prop := in : Prop → Pb.
```

Potentially problematic because $PB : \mathbf{Prop}$ but PB intuitively isomorphic to **Prop**.

Restrictions of elimination depending on sorts

Elimination rule for type *bool* (all sorts possible)

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

Elimination rule for the type *or A B* (only on **Prop**)

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

Rules on the sorts for the elimination

- The elimination of inductive types in **Type** (predicative hierarchy) has no restriction (*weak elimination – towards **Prop** and **Set** – and strong – towards **Type***)
- Elimination of inductive types in **Prop** is restricted :
  - in general, one cannot build a type in **Type** by case on the proof-term in a proposition according to the implicit interpretation of **Prop** as proof-irrelevant (*propositional elimination only*)

    ```
    fun (p:or A B) ⇒ match p with
          (or_introl a) ⇒ true | (or_intror b) ⇒ false
                        end.
    ```

  - exception Singleton types : if the type in **Prop** has zero constructor (absurdity) or a unique constructor whose arguments are in **Prop** (equality, conjunction . . .).
    We allow *weak and strong elimination*
  - partial exception : if the type in **Prop** has a unique constructor which arguments are either propositions of type **Prop** *or* small arities (type schemes which build in **Prop**), then elimination towards **Set** is allowed (*weak elimination – only towards small types –*)

In pratice in Coq

For each inductive definition of a type I, Coq defines automatically associated elimination schemes (when allowed)

- strong elimination (to **Type**) : *I_rect*
- elimination to small computational types (to **Set**) : *I_rec*
- elimination to logical propositions (to **Prop**) : *I_ind*

Moreover, by default, eliminations are dependent when I is computational (in **Set** or **Type**) and non-dependent when in **Prop**.

Examples

```
Inductive True : Prop := I : True.
True_rect : ∀P : Type, P → True → P
True_rec : ∀P : Set, P → True → P
True_ind : ∀P : Prop, P → True → P

Inductive unit : Type := tt : unit.
unit_rect : ∀P : unit → Type, P tt → ∀u : unit, P u
unit_rec : ∀P : unit → Set, P tt → ∀u : unit, P u
unit_ind : ∀P : unit → Prop, P tt → ∀u : unit, P u
```

To generate schemes which are not automatically generated, one can use the command `Scheme`. Example :

```
Scheme True_indd := Induction for True Sort Prop.
True_indd
     : ∀P : True → Prop, P I → ∀t : True, P t
```

Strong elimination

- Possibility to build a proposition or a type by case analysis or recursion.
- Proof of $\texttt{true} \neq \texttt{false}$

```
Inductive False : Prop :=.
Definition P (b: bool) : Prop := match b with
  true ⇒ True | false ⇒ False end
```

$$\frac{\texttt{true} = \texttt{false} \quad P(\texttt{true}) \equiv \texttt{True}}{P(\texttt{false}) \equiv \texttt{False}}$$

Summary

Introduction to the Calculus of Inductive Constructions
Proof Assistants
From the Calculus of Constructions to the Calculus of Inductive Constructions
Examples of inductive definitions

Specifics of the Calculus of Inductive Constructions
Fixpoint operators
Conditions for inductive definitions
Advanced inductive definitions

Inductive definitions with internal dependencies

```
Inductive ex (A:Type) (P:A → Prop) : Prop :=
  ex_intro : ∀x:A, P x → ex (A:=A) P.
```

Can we project on first and second components ?

```
Inductive sigT (A:Type) (P:A → Type) : Type :=
    existT : ∀x:A, P x → sigT P.
```

Can we project on first and second components ?

Higher-order inductive definitions

Example of Kleene's recursive ordinals.

```
Inductive ord : Type :=
| O : ord
| S : ord → ord
| lim : (nat → ord) → ord
```

Induction schems (Coq syntax)

```
fun (P:ord→Type)(f:P O) (f0:∀o : ord, P o → P (S o))
    (f1 : ∀o:nat→ord, (∀n:nat,P (o n)) → P (lim o)) ⇒
fix F (o : ord) : P o :=
  match o as o0 return (P o0) with
  | O ⇒ f
  | S o0 ⇒ f0 o0 (F o0)
  | lim o0 ⇒ f1 o0 (fun n : nat ⇒ F (o0 n))
  end
 : ∀P : ord → Type,
    P O → (∀o:ord, P o→P (S o)) →
    (∀o:nat→ord,(∀n:nat,P (o n)) → P (lim o)) →
    ∀o:ord, P o
```

Dependent inductive definitions : example of equality

```
Inductive eq (A:Type) (x:A) : A → Prop :=
  refl_equal : eq A x x.
```

- a family of inductive types

$$\frac{}{\Gamma \vdash eq : \forall A : \mathbf{Type}, A \to A \to \mathbf{Prop}}$$

  - the first two parameters are “family” parameters
  - the third one is an “index”
- elimination rule without dependency with the filtered term : rewriting !

$$\frac{\Gamma \vdash t : eq\,A\,a\,b \quad \Gamma, c : A \vdash A(c) : s \quad \Gamma \vdash u : A(a)}{\Gamma \vdash \left(\begin{array}{l}\texttt{match}\ t\ \texttt{in}\ eq\ \_\ \_\ c\ \texttt{return}\ A(c)\ \texttt{with}\\ \quad refl\_equal \Rightarrow u\\ \texttt{end}\end{array}\right) : A(b)}$$

Remark : elimination on all sorts because equality is a singleton type

Mutual inductive definitions : example of forests and trees

```
Inductive tree (A:Type) : Type :=
  | node : A → (forest A) → (tree A)
with forest (A:Type) : Type :=
  | empty : (forest A)
  | add : (tree A) → (forest A) → (forest A).
```

Can be simulated by :

```
Inductive tree_for (A:Type) : bool → Type :=
  | node : A → tree_for A false → tree_for A true
  | empty : tree_for A false
  | add : tree_for A true → tree_for A false
          → tree_for A false.

Definition tree (A:Type) := tree_for A true.
Definition forest (A:Type) := tree_for A false.
```

Mutually inductive definitions : example of forests and trees

```
Inductive tree (A:Type) : Type :=
  | node : A → (forest A) → (tree A)
with forest (A:Type) : Type :=
  | empty : (forest A)
  | add : (tree A) → (forest A) → (forest A).
```

Can also be simulated by

```
Inductive tree_aux (A:Type) (forest:Type): Type :=
  | node : A → forest → tree A forest.

Inductive forest (A:Type) : Type :=
  | empty : (forest A)
  | add : tree_aux A (forest A) → forest A → forest A.

Definition tree (A:Type) := tree_aux A (forest A).
```

When mutually inductive definitions are in different sorts, only the second encoding is possible. It requires an extended strict positivity condition which allows imbricated definitions.

Mutual fixpoints : example of the size of a forest

```
Definition tree_size := fun (A:Type) ⇒
  fix tree_size (t:tree A) : nat :=
    match t with
    | node A f ⇒ S (forest_size f)
    end
  with forest_size (f:forest A) : nat :=
    match f with
    | empty ⇒ 0
    | add t f' ⇒ tree_size t + forest_size f'
    end
  for tree_size.
```

Fixpoints with parameters

A fixpoint in the Calculus of Inductive Constructions may have several arguments.

```
Inductive vect : nat → Type :=
| vnil : vect O
| vcons : ∀n, nat → vect n → vect (S n).

Definition sum :=
  fix sum (n:nat) (ln:vect n) {struct ln} : nat :=
    match ln return nat with
    | vnil ⇒ O
    | vcons n' p ln' ⇒ p + sum n' ln'
    end.
```

We use the notation `{struct` *x*`}` structurally decreasing argument.

Dependent inductive definitions : example of accessibility

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

$Acc\ A\ R\ x$ expresses that any decreasing (following R) chain from x is well-founded.
$\forall x, Acc\ A\ R\ x$ expresses that R is a well-founded relation in A.

Non structural decreasing

Acc is the natural tool to transform any well-founded relation into a structural order. A fonction $f(x)$ prouvably terminating through a well-founded order $\leq$ can be defined by

```
fix msort (l:list nat)(H:Acc le (length l)){struct H}
 : list nat :=
  match H with Acc n Hn ⇒
  ..msort l1 (Hn (length l1) (* proof of |l1|<|l| *)..
  ..msort l2 (Hn (length l2) (* proof of |l1|<|l| *)..
  end.
```

One actually writes

```
msort l1
    (match H with
      Acc n Hn ⇒ Hn (length l1) (* proof of |l1|<|l| *)
     end)
```

Non structural termination

Coq has a macro for doing that : `Function`.

```
Definition R (l1 l2:list nat) := length l1 < length l2.

Function msort (l:list nat) {wf R l} : list nat :=
  match H with Acc n Hn ⇒
  ..msort l1 (Hn (length l1) (* proof of |l1|<|l| *)..
  ..msort l2 (Hn (length l2) (* proof of |l1|<|l| *)..
  end.
```

Parameters recursively non uniform

Coq 8.1 allows parameters which are recursively non uniform. So one can rewrite *Acc* as

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

Dependent Inductive definitions : example

```
Inductive prove : list formula → formula → Prop :=
| ProofImplyE : ∀A B Gamma,
     Gamma |- (A -→ B) → Gamma |- A → Gamma |- B
| ProofImplyI : ∀A B Gamma,
     (A::Gamma) |- B → Gamma |- (A -→ B)
| ProofAx : ∀A Gamma C, In A Gamma → Gamma |- A

where "Gamma |- A" := (prove Gamma A).
```

equivalent to

```
Inductive prove (Gamma:list formula)(C:formula) :Prop :=
| ProofImplyE
   : ∀A, Gamma |- (A-→C) → Gamma |- A → Gamma |- C
| ProofImplyI
   : ∀A B, C=A-→B → (A::Gamma) |- B → Gamma |- C)
| ProofAx : In C Gamma → Gamma |- C

where "Gamma |- A" := (prove Gamma A).
```

Inversion

Inversion principle

```
prove Gamma C →
(∃A, ∃B, C=A-→B ∧ prove (A::Gamma) B) ∨
(∃A, prove Gamma (A -→ B) ∧ prove Gamma A) ∨
(In C Gamma)
```

Free if we choose a fully parameterized definition.

Coinductive types

```
Inductive Stream : Set
  := Cons : A → Stream → Stream.
```

This type is empty

```
Fixpoint empty (s:Stream A) : False :=
  match s with (Cons _ t) ⇒ empty t end

CoInductive Stream : Set
  := Cons : A → Stream → Stream.

CoFixpoint zeros : Stream nat := Cons 0 zeros.
CoFixpoint from (n:nat) : Stream nat
   := Cons n (from (S n)).
```

Guard conditions : recursive calls protected by a constructor.