Inductive Types

Part one :

Computation and Deduction

with chunks of Coq in it !!

Benjamin Werner

INRIA–Rocquencourt

Proofs-as-Programs Summer School

Eugene, Oregon, june 27^{th} 2002

Prologue 0 : logical cuts

$$\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \qquad \dfrac{\pi_2}{\Gamma \vdash B}}{\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\wedge\text{-e1}}\wedge\text{-i}$$

simplifies to

$$\dfrac{\pi_1}{\Gamma \vdash A}$$

Prologue 1 : numbers in Arithmetic

Axioms :

$$\begin{aligned}
\forall x \ . \ 0 + x &= x \\
\forall xy \ . \ S(x) + y &= S(x + y) \\
\forall x \ . \ 0 * x &= 0 \\
\forall xy \ . \ S(x) * y &= y + x * y
\end{aligned}$$

Induction scheme :

$$P(0) \to (\forall x \ . \ P(x) \to P(S(x))) \to \forall n \ . \ P(n)$$

Prologue 2 : axiomatic cuts

$$\dfrac{\dfrac{\pi_0}{P(0)} \qquad \dfrac{\pi_S}{\forall n.P(n) \Rightarrow P(S(n))}}{\dfrac{\forall n.P(n)}{P(0)}}$$

simplifies to

$$\dfrac{\pi_0}{P(0)}$$

Real axioms deserve real cuts

Digression : Coq Syntax crash course

$\lambda x : A.t$ `[x:A]t`

$\forall x : A.t$

$\Pi x : A.B$ `(x:A)B`

What is a Type ?

A type is a Set :

$\mathbb{N}$, $\mathbb{N} \to \mathbb{N}$ are types

$\{n : \mathbb{N} \mid even(n)\}$ can be a type

$\{0 \ ; \ x \mapsto x + 1\}$ is not

A type is a set with a uniformity of structure. This allows uniform definition of programs.

Concrete Types (in programming)

In ML (SML, Caml. . .), a concrete type is defined as closed by a set of constructors.

```
type bool = true | false


type nat = O | S of nat



let bnot = function true  -> false
                  | false -> true
```

Adding concrete types to your logic

Coq syntax :

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

Inductive nat  : Set :=  O : nat | S : nat->nat.

Fixpoint plus [n,m:nat] : nat :=
  Cases n of  O => m
           | (S p) => (S (plus p m))
  end.
```

New reductions $\Rightarrow$ new proofs ! ! !

$$(\text{plus } O \ m) \ \rhd \ m$$

$$(\text{plus } (S \ n) \ m) \ \rhd \ (S \ (\text{plus } n \ m))$$

plus is a program : $(\text{plus } 2 \ 2) \rhd 4$

This extends to propositions :

$$(\text{plus } 2 \ 2) = 4 \ \rhd \ 4 = 4.$$

(conv) $\dfrac{\Gamma \vdash t : A}{\Gamma \vdash t : B} A =_\beta B$

2+2=4 is proved by reflexivity !

reasoning about inductive types

"Inductive" means the type is the smallest type closed under the constructors.

The only canonical objects of type `nat` are `O`, `(S O)`, `(S (S O))`, etc

Given any property $P : \mathrm{nat} \rightarrow \mathrm{Prop}$, if :

– $(P\ O)$ holds,

– $\forall n : \mathrm{nat}. (P\ n) \rightarrow (P\ (S\ n))$,

then $(P\ n)$ holds for any $n : \mathrm{nat}$

$\Rightarrow$ induction principle

reasoning about inductive types

For any inductive type definition, we add an induction axiom

```
nat_ind
     : (P:(nat->Prop))
         (P O)->
         ((n:nat)(P n)->(P (S n)))->
          (n:nat)(P n)
```

Same thing for booleans, lists, etc.

termination is an issue

Any (closed) object of type nat reduces either to O or $(S\ x)$.

$S^\omega = (S\ (S\ (S\ (S \ldots$ is not a natural number (it contradicts induction)

`Fixpoint foo [n:nat] : nat := (foo n).`

`(foo O)` is not a number

termination of computations is necessary to enforce the induction principle

structural recursion

Answer : we restrict recursive calls to structuraly smaller arguments.

```
Fixpoint plus [n,m:nat] : nat :=
  Cases n of  O => m
            | (S p) => (S (plus p m))
  end.
```

positivity condition

Inductive Foo : Set := C : (~~Foo~~->Foo)->Foo.

is problematic :

– What should the induction principle be ?

– It breaks the termination property even without Fixpoint command !

in Caml :

```
type foo = C of (foo->foo)
let loop (C f) = f(C f)
```

then (loop (C loop)) $\triangleright$ (loop (C loop))

To sum up

To first, or higher, -order logic, we add datatypes à la ML,

with positivity restriction,

with structural recursion restriction,

with a *ad hoc* induction axiom for each definition.

we obtain :

a logic where objects are programs

shorter proofs through computations.

Gödel's system T

It is the core of this calculus.

I.e. simply-typed λ-calculus extended with :

an atomic type $\mathsf{nat}, O : \mathsf{nat}, S : \mathsf{nat} \to \mathsf{nat}$

a family of combinators

$R_T : T \to (\mathsf{nat} \to T \to T) \to \mathsf{nat} \to T$

reduction rules :

$$
\begin{array}{rcl}
(R_T\ t_0\ t_S\ O) & \rhd & t_0 \\
(R_T\ t_0\ t_S\ (S\ n)) & \rhd & (t_S\ n\ (R_T\ t_0\ t_S\ n))
\end{array}
$$

Theorem System T is strongly normalizing and confluent

Proof : Usual reducibility technique.

Informal : system T moraly is the system I described up to here.

Induction Cuts

let :

`P:nat->Prop  p0:(P O)  pS:(n:nat)(P n)->(P(S n))`

then

`(nat_ind P p0 pS O) : (P O)` should simplify to `p0`

`(nat_ind P p0 pS (S n))` should simplify to `(pS n (nat_ind P p0 pS n))`

Hey ! these are exactly the reductions of system T. . .

Martin-Löf's Type Theory (1)

Martin-Löf's Type Theory is :

System T enriched with dependent types

or equivalently :

$\lambda\Pi$ enriched with inductive types

New rules :

$[] \vdash \mathsf{nat} : \mathsf{Prop} \qquad [] \vdash O : \mathsf{nat} \qquad [] \vdash S : \mathsf{nat} \to \mathsf{nat}$

$$\frac{\Gamma \vdash T : \mathsf{nat} \to \mathsf{Prop}}{R_T : (P\ \ O) \to (\forall m : \mathsf{nat}.(P\ \ m) \to (P\ \ (S\ \ m))) \to \forall n : nat.(P\ \ n)}$$

This dependent typing of R_T can be obtained by generalizing the typing of pattern matching.

```
P:nat->Prop
p0 : (P O)
pS : (n:nat)(P (S n))
x:nat

<P>Cases x of O => p0
            | (S n) =>(pS n) end : (P x)
```

Inductive definitions of connectors

$$\frac{\Gamma \vdash A \wedge B \quad \Gamma \vdash A \to B \to C}{\Gamma \vdash C} \ (\wedge\text{-e})$$

coded as an inductive definition !

```
Inductive and : Prop := conj : A -> B -> and.
```

Inductive definition of disjunction

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\frac{\Gamma \vdash A \rightarrow C \quad \Gamma \vdash B \rightarrow C \quad \Gamma \vdash A \vee B}{\Gamma \vdash C}$$

```
Inductive or [A,B:Prop] : Prop :=
  left : A-> (or A B)
| right : B -> (or A B).
```

The existential quantifier

Martin-Lof presentation :

$$\frac{\Gamma \vdash A : Prop \quad \Gamma, x : A \vdash B : Prop}{\Gamma \vdash \Sigma x : A.B : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x \setminus a]}{\Gamma \vdash (a, b) : \Sigma x : A.B}$$

$$\frac{\Gamma \vdash c : \Sigma x : A.B}{\pi_1(c) : A} \qquad \frac{\Gamma \vdash c : \Sigma x : A.B}{\pi_2(c) : B[x \setminus \pi_1(c)]}$$

$$\pi_1(a, b) \rhd a \qquad \pi_2(a, b) \rhd b$$

Retrieving Heyting's semantics

I an inductive type

$[] \vdash t : I$

Then the normal form of t starts with a constructor.

$I = \Sigma x : A.B \Rightarrow t \rhd^* (a, b)$

$I = A \vee B \Rightarrow t \rhd^* left(a)$ or $right(b)$

Inductive predicates

The smallest set :

– containing 0

– closed by $x \mapsto x + 2$

```
Inductive even : nat -> Prop :=
  E0 : (even O)
| ES : (n:nat)(even n)->(even (S (S n))).
```

$\Rightarrow$ associated induction principle

Why not the impredicative encoding ?

- induction scheme is not provable in CC
- slow computations
- extraction towards ML
- proving $0 \neq 1$

Conclusion

Smooth materialization of Curry-Howard

Using Computation in Proofs

Computing with proofs

Powerfull generic mechanism