

Inductive Types

Part two :

Advanced features

Benjamin Werner

INRIA-Rocquencourt

Proofs-as-Programs Summer School
Eugene, Oregon, June 27th 2002

Principle : each constructor describes one possible **canonical form** of the inhabitants of the inductive type.

Example :

Inductive and $[A, B : \text{Prop}] : = \text{conj} : A \rightarrow B \rightarrow (\text{and } A B$

a canonical $c : (\text{or } A B)$ is of the form

$c = (\text{conj } a b)$ with $a : A$ and $b : B$.

Example 2 :

Inductive or [A,B:Prop] :=

or_intro1 : A->(or A B)

| or_intro2 : B->(or A B) .

two possible canonical forms :

(or_intro1 a) with a : A

(or_intro2 b) with b : B

Example 3 (with dependent types) :

```

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

one possible canonical form/proof :
  (ex_intro a p)
with a : A and p : (P a) .

```

Inductive predicates

The smallest set :

– containing 0

– closed by $x \mapsto x + 2$

Inductive even : nat -> Prop :=

E0 : (even 0)

| ES : (n:nat) (even n) -> (even (S n)).

\Leftrightarrow associated induction principle

If $(P\ 0)$ and $(P\ n) \rightarrow (P\ (S\ n))$ then
 $(P\ n)$ holds for every even number n .

`even_ind`

`: (P : nat → Prop)`

`(P 0)`

`→ ((n : nat) (even n) → (P n) → (P (S n)))`
`→ (n : nat) (even n) → (P n)`

Defining "less or equal"

Given n , we define inductively the set of numbers greater or equal to n :

- it contains n
- it is closed under successor

```
Inductive le [n : nat] : nat -> Prop :=  
  le_n : (le n n)  
  | le_s : (m:nat) (le n m) -> (le n (S m)) .
```

Notice the use of parameter.

canonical proofs

less-or-equal induction principle

If $(P\ n)$ and P is closed for the successor,
 then P is true for all number
 greater-or-equal to n

`le_ind`

`: (n:nat; P:(nat->Prop))`

`(P n)`

`->((m:nat) (le n m)->(P m)->(P (S m)))`
`->(x:nat) (le n x)->(P x)`

Reminder : lists of numbers

```
Inductive list : Set :=  
  nil : list  
| cons : nat -> list -> list.
```

Inductive predicates as Prolog specs

List concatenation specified as a ternary predicate : (concat l1 l2 l3) iff l3=l1@l2

Inductive concat :

con_nil : (l2:list)(concat nil l2 l2)

| con_cons :

(l1,l2,l3:list)(concat l1 l2 l3)->

(a:nat)

(concat (cons a l1) l2 (cons a l3)).

Prolog program with type discipline

Compiling prolog

The ML program :

Given I_1 and I_2 it computes I_3 such that
(concat I_1 I_2 I_3)

We can state the prolog program always
succeeds :

($I_1, I_2: \text{list}()$ (EX I_3 | (concat I_1 I_2 I_3))

We can prove this statement :

p-con : ($I_1, I_2: \text{list}()$ (EX I_3 | (concat I_1 I_2 I_3))

Compiling prolog - 2

```
p-con : (l1,l2:list) (EX l3 | (concat l1 l2 l3))
```

The term p-con :

– takes l1 and l2 as arguments

– returns a list l3 together with a proof

that (concat l1 l2 l3).

so p-con is a **compiled version** of the

concat prolog program (in "ML").

With program extraction, we can refine to

```
p-con' : list->list->list.
```

Types Defined by Recursion

Proving $0 \neq 1$

True propositions, false propositions

A trivial true proposition :

Inductive True : Prop := I : True.

An obviously false proposition :

Definition False := (P:Prop)P.

Attention ! True and False are **not** true
and false.

They are propositions and not booleans !

Proving $0 \neq 1$

$0 \neq 1$ stands for $0 = 1 \rightarrow \text{False}$

$\text{False} : \text{Prop}$ is $(P : \text{Prop}) \rightarrow \text{False} \rightarrow P$

$0 = (S 0)$ stands for :

$(P : \text{nat} \rightarrow \text{Prop}) (P 0) \rightarrow (P (S 0))$

Proving $0 \neq 1$ is finding $P : \text{nat} \rightarrow \text{Prop}$ such

that :

– $(P 0)$ is provable

– $(P (S 0))$ implies False .

How to define P ?

Solution : allowing the definition of propositions / types by case analysis.

Definition P : nat->Prop :=

[n:nat]cases n of

0 => True

| (S p) => False

end.

This is possible by relaxing the typing rule for case analysis : we allow **types** to appear on the right.

We now have the reductions :

$(P\ 0) \triangleright \text{True}$ thus $(P\ 0) \langle - \rangle \text{True}$

$(P\ (S\ 0)) \triangleright \text{False}$ thus $(P\ (S\ 0)) \langle - \rangle \text{False}$

Therefore, if $0=(S\ 0)$, we have $\text{True} \langle - \rangle \text{False}$.

So $0=(S\ 0) \langle - \rangle \text{False}$.

The proof term is : $[e:0=(S\ 0)](e\ P\ I)$

indeed :

$(e\ P) : (P\ 0) \langle - \rangle (P\ (S\ 0)) =_{\beta} \text{True} \langle - \rangle \text{False}$

and $I : \text{True}$

Specifying and certifying a
sorting program

Inductive list : Set :=

nil : list

| cons : nat->list->list.

We want to sort these lists in increasing
order

sorted lists – Inductive version

```
Inductive low [a:nat] : list -> Prop :=  
  low_nil : (low a nil)  
  | low_cons : (l:list)(b:nat)(le a b)->  
    (low a (cons b l)).
```

```
Induction sorted : list -> Prop :=  
  sorted_nil : (sorted nil)  
  | sort_cons : (l:list)(a:nat)(sorted l)->  
    (low a l)->(sorted (cons a l)).
```

Can we prove : (sorted (cons a l))->(sorted l) ?

sorted lists – stand-alone recursive version

```
Definition low' := [a:nat ; l : list]
  Cases l of
  nil => True
  | (cons b _) => (le a b)
end.
```

```
Fixpoint sorted' [a:nat ; l:list] : Prop :=
  Cases l of
  nil => True
  | (cons b m) => (low b m) /\ (sorted' m)
end.
```

List permutations

```
Inductive perm : list -> list -> Prop :=
| perm_refl : (l:list)(perm l l)
| perm_trans : (l1,l2,l3:list)(perm l1 l2 -> perm l2 l3 -> perm l1 l3)
| perm_hd : (a,b:nat)(l:list)
  (perm (cons a l))
  (perm (cons b (cons a l)))
| perm_tl : (a:nat)(l1,l2:list)(perm l1 l2 ->
  (perm (cons a l1)(cons a l2)))
```

specification of the sorting program

$$\forall l : \text{list} . \exists l' : \text{list} . \text{sorted}(l') \wedge \text{permut}(l, l')$$
$$(\exists l : \text{list} . \text{sorted } l) \wedge (\exists l' : \text{list} . \text{sorted } l') \wedge \text{permut } l \ l'$$

A (constructive) proof of this proposition

contains the sorting program.

One proof for quicksort, one proof for

heapsort, etc.

Program Extraction principle

`(l:list)(EX l' : list | (sorted l')/\(permut l l'`
Mark the computational content.

extraction of a program `list->list`