Inductive Types

Part 3-1 :
Generalizations and definitions

Benjamin Werner

INRIA–Rocquencourt

Proofs-as-Programs Summer School

Eugene, Oregon, july 1^{st} 2002

General Form of Inductive Definitions

$I \quad : \quad (\overline{x_i : A_i})s$ the inductive type/predicate

$C_j \quad : \quad (\overline{y_k^j : T_k^j})(I \ \overline{a_i^k})$ type of each constructor

positivity :

– either $I \notin T_k^j$, or

– $T_k^j = (\overline{z_l : U_l})(I \ \bar{b})$ with $I \notin U_l$

Examples :

cons : $A \to$ list $\to$ list , left : $A \to (or \ A \ B)$

Beyond data-types

```
Inductive ord : Set :=
  Oo : ord
| So : ord -> ord
| lim : (nat->ord) -> ord.
```

`(lim f)` is a canonical “ordinal”.

For any `n:nat`, `(f n)` is structuraly smaller than `(lim f)`.

These are infinitely branching trees,

they cannot be (finitely) printed.

Even further

```
Inductive Ens : Type :=
  sup : (A:Type)(A->Ens)->Ens.
```

Here we even branch w.r.t. an arbitrary type!

Not very intuitive...

Very powerful : this type encodes sets of Zermelo set theory

Wait for Alexandre's lectures

Restrictions w.r.t. sorts

Consider :

```
Inductive capture : Set :=
  c_i : Set -> capture.
```

This definition is correct

But if we allow projection :

`proj : capture -> Set` with

`(proj (c_i A))`$\triangleright$`A`

then we can "encode" `Set:Set`

The reason is that `c_1` quantifies over Sets.

$\Rightarrow$ elimination towards `Set` is forbidden for such types.

The real typing of `Ens` :

```
Inductive Ens : Type(i+1):=
  sup : (A:Type(i))(A->Ens)->Ens.
```

The sorts `Set` and `Prop`

They are "twins"

The "Prop part" can be erased for a realisability interpretation.

For example, Harrop formulas should be of type `Prop`.

We should not use `Prop` terms to compute `Set` terms.

Program extraction 1 : dividing by two

```
Fixpoint D [n:nat] : nat :=
Cases n of O => O
    | (S p) => (S(S(D p)))
end.
Inductive even : nat -> Prop :=
  e0 : (even O)
| eS : (n:nat)(even n)->(even (S(S n))).
```

Two existentials

```
(EX p:nat | n=(D p)) : Prop
{ p:nat | n = (D p)} : Set
```

```
(n:nat)(Even n)->(EX p:nat | n=(D p)) : Prop
(n:nat)(Even n)->{ p:nat | n = (D p)} : Set
```

First one can be proved by induction over the proof of `(even n)`, the second one cannot.

```
let rec ins l a x =
  match l with
    nil -> cons (a, nil)
  | cons (n, l0) ->
       (match Compare_dec.le_ge_dec a n with
          left -> cons (a, (cons (n, l0)))
        | right -> cons (n, (ins l0 a prop)))
```