MRI Master Class
Computer Mathematics
Nijmegen NL, 2003

Type Systems

Herman Geuvers
Nijmegen University, NL

Lecture 6: Inductive Types

Inductive Types by Example

Inductive types or maybe better inductive definitions in Coq.

"`Coq <`" is the Coq prompt.

```
Coq < Inductive nat : Set :=
Coq <       O:nat
Coq <     | S:nat->nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined
```

This generates, besides the constants `nat_ind`, `nat_rec` and `nat_rect` named above, also the constants

```
O : nat
S: nat -> nat
```

The types of the other constants can be asked (by the command Check).

```
nat : Set
nat_ind
  : (P:nat->Prop)(P O) -> ((n:nat)(P n)->(P (S n)))
                  -> (n:nat)(P n)
nat_rec
  : (P:nat->Set)(P O) -> ((n:nat)(P n)->(P (S n)))
                  -> (n:nat)(P n)
nat_rect
  : (P:nat->Type)(P O) -> ((n:nat)(P n)->(P (S n)))
                  -> (n:nat)(P n)
```

Meaning and use of `nat_ind`, `nat_rec`, `nat_rect`.

`nat` should represent the smallest set closed under the two operations `0` and `S`.

- Everything constructed using `0` and `S` is of type `nat`.
  This is secured by `0` : `nat` and `S` : `nat`→`nat`.
- The terms constructed using `0` and `S` are the only closed terms of type `nat`. (no junk)
- The closed terms t and t' of type `nat` are equal iff t and t' are exactly the same term. (no additional equalities, free algebra)

This second and third property are secured by `nat_ind`, `nat_rec` and `nat_rect`.

For the inductive type nat,

- the terms 0 and S are also called the constructors
- the terms nat_ind, nat_rec, nat_rect are also called the eliminators.

Their typing can also be given by deduction rules: introduction and elimination rules

(intro) $\dfrac{}{\vdash \mathtt{0} : \mathtt{nat}} \quad \dfrac{}{\vdash \mathtt{S} : \mathtt{nat} \rightarrow \mathtt{nat}}$

(elim) $\dfrac{\Gamma \vdash P : \mathtt{nat} \rightarrow \mathtt{Prop} \quad \Gamma \vdash f_0 : P\mathtt{0} \quad \Gamma \vdash f_1 : \Pi x{:}\mathtt{nat}.Px \rightarrow P(\mathtt{S}x)}{\vdash \mathtt{nat\_ind}\, f_0 f_1 : \Pi x{:}\mathtt{nat}.Px}$

(elim) $\dfrac{\Gamma \vdash P : \mathtt{nat} \rightarrow \mathtt{Set} \quad \Gamma \vdash f_0 : P\mathtt{0} \quad \Gamma \vdash f_1 : \Pi x{:}\mathtt{nat}.Px \rightarrow P(\mathtt{S}x)}{\vdash \mathtt{nat\_rec}\, f_0 f_1 : \Pi x{:}\mathtt{nat}.Px}$

(elim) $\dfrac{\Gamma \vdash P : \mathtt{nat} \rightarrow \mathtt{Type} \quad \Gamma \vdash f_0 : P\mathtt{0} \quad \Gamma \vdash f_1 : \Pi x{:}\mathtt{nat}.Px \rightarrow P(\mathtt{S}x)}{\vdash \mathtt{nat\_rect}\, f_0 f_1 : \Pi x{:}\mathtt{nat}.Px}$

Computation

The reduction rules associated to these constructors and eliminators are the following.

$$\begin{aligned}\mathtt{nat\_rec}\, f_0 f_1 \mathtt{0} &\longrightarrow_\iota f_0,\\ \mathtt{nat\_rec}\, f_0 f_1 (\mathtt{S}x) &\longrightarrow_\iota f_1 x (\mathtt{nat\_rec}\, f_0 f_1 x),\end{aligned}$$

etc. for $\mathtt{nat\_ind}, \mathtt{nat\_rect}$

These reduction rules express the usual rule that an introduction which is immediately followed by an elimination can be reduced.
This computation gives higher order primitive recursion:
Given $f_0 : A$, $f_1 : \mathtt{nat}{\rightarrow}A{\rightarrow}A$, define

$$h : \mathtt{nat}{\rightarrow}A \text{ by } h := \mathtt{nat\_rec}\, f_0 f_1$$

Then

$$\begin{aligned} h\mathtt{0} &=_{\beta\iota} f_0,\\ h(\mathtt{S}x) &=_{\beta\iota} x \;\; f_1\, x\, (hx).\end{aligned}$$

Examples

1. The eliminator `nat_ind` is used for proofs by induction.

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

If $P : \texttt{nat} \to \texttt{Prop}$ is a property of natural numbers, then we can prove $\forall n{:}\texttt{nat}.Pn$ by proving

- $P\texttt{O}$ (the base case) and
- $\forall n{:}\texttt{nat}.Pn \to P(\texttt{S}n)$ (the step-case).

2. The eliminator `nat_rec` is used for defining functions by primitive recursion.

```
nat_rec
   : (P:nat->Set)(P O) -> ((n:nat)(P n)->(P (S n)))
                         -> (n:nat)(P n)
```

Example: the plus-function of type $\mathtt{nat}{\to}\mathtt{nat}{\to}\mathtt{nat}$. In the elimination rule for `nat_rec` above, take

$$
\begin{aligned}
P &:\equiv \lambda n{:}\mathtt{nat}.\mathtt{nat}{\to}\mathtt{nat}(:\mathtt{nat}{\to}\mathtt{Set}),\\
f_0 &:\equiv \lambda n{:}\mathtt{nat}.n(:\mathtt{nat}{\to}\mathtt{nat}),\\
f_1 &:\equiv \lambda x{:}\mathtt{nat}.\lambda g{:}\mathtt{nat}{\to}\mathtt{nat}.\lambda n{:}\mathtt{nat}.\mathtt{S}(gn)(:\Pi x{:}\mathtt{nat}.Px{\to}P(\mathtt{S}x)).
\end{aligned}
$$

Define $\mathrm{plus} :\equiv \mathtt{nat\_rec} f_0 f_1$. Then indeed

$$
\begin{aligned}
\mathrm{plus}\mathtt{O}y &\longrightarrow_\iota f_0 y \twoheadrightarrow_\beta y,\\
\mathrm{plus}(\mathtt{S}x)y &\longrightarrow_\iota f_1 x(\mathrm{plus}x)y \twoheadrightarrow_\beta \mathtt{S}(\mathrm{plus}xy).
\end{aligned}
$$

3. The eliminator nat_rect is used for defining predicates by induction. As an example define the predicate Odd:nat→Prop. In the elimination rule for nat_rect, take

$$\begin{aligned} P &:\equiv \lambda n{:}\mathtt{nat}.\mathtt{Prop}, \\ f_0 &:\equiv \mathtt{False}, \\ f_1 &:\equiv \lambda x{:}\mathtt{nat}\lambda\alpha{:}\mathtt{Prop}.\neg\alpha. \end{aligned}$$

Define $\mathrm{Odd} :\equiv \mathtt{nat\_rect}\, f_0 f_1$, then

$$\begin{aligned} \mathrm{Odd}\mathtt{0} &\longrightarrow_\iota \mathtt{False}, \\ \mathrm{Odd}(\mathtt{S}x) &\longrightarrow_\iota f_1 x(\mathrm{Odd}x) \twoheadrightarrow_\beta \neg(\mathrm{Odd}x). \end{aligned}$$

Syntactic sugar for defining higher order primitive recursive functions in Coq: definitions by cases (pattern matching) and well-founded recursion. (NB: Plus now defined by rec. over 2nd arg.)

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

This is now the primitive definition in Coq. "`Print plus.`"

```
Fix plus
  {plus [n:nat; m:nat] : nat :=
     Cases m of
       O => n
     | (S p) => (S (plus n p))
     end}
```

Example of Fibonacci function

(NB: the notation "`(1)`" can be used to denote "`(S O)`")

```
Fixpoint nfib [n:nat] : nat :=
Cases n of
 | O     => (1)
 | (S m) =>
        Cases m of
         | O     => (1)
         | (S p) => (plus (plus (nfib p)(nfib m)) (1))
        end
end.
```

Inductive type of lists

There are (at least) three versions of the well-known inductive type of lists.

(1) Simplest variant: lists over natural numbers.

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

Elimination principle for elimination ‘over `Prop`’.

```
list_ind
: (P:list->Prop)
    (P nil) -> ((n:nat)(l:list)(P l)->(P (cons n l)))
       -> (l:list)(P l)
```

(2) Another option: define the type of parametrized lists, parametrized over the type of its elements.
Define `List` : `Set`→`Set`, as an inductively defined function.

```
Inductive List: (A:Set)Set :=
Nil : (A:Set)(List A)
| Cons : (A:Set)A -> (List A) -> (List A).
```

In the elimination principle we have to abstract over A:`Set`: the quantification is not over P : (`List` A)→`Prop`, but over $P : \Pi A$:`Set`.(`List` A)→`Prop`.

```
List_ind : (P:(A:Set)(List A)->Prop)
((A:Set)(P A (Nil A)))
 -> ((A:Set)(y:A)(l:(List A))(P A l)->(P A (Cons A y l)))
    -> (A:Set)(l:(List A))(P A l)
```

Or, with some more sugar:

```
List_ind : (P:((A:Set)(List A)->Prop))
((A:Set)(P ? (Nil ?)))
 -> ((A:Set; a:A; l:(List A))(P ? l)->(P ? (Cons ? a l)))
    -> (A:Set; l:(List A))(P ? l).
```

- This definition is convenient for properties of lists that do not refer to the internal structure of the element-type A (generic properties). (E.g. 'being an empty list' or 'being a list of length n'.)
- A predicate that deals with the internal structure of the element-type A can not be handled in this way. (E.g. if the element type is `nat`, the predicate 'being a list with maximum element n' is not a generic property of lists.)

(3) Yet another approach, that treats the element type as a real parameter.

```
Inductive LIST [A:Set] :Set :=
NIL : (LIST A)
| CONS : A -> (LIST A) -> (LIST A).

LIST_ind : (A:Set) (P:(LIST A)->Prop)
 (P (NIL A))
  -> ((y:A)(l:(LIST A))(P l)->(P (CONS A y l)))
     -> (l:(LIST A))(P l)
```

- This induction principle works for every individual A:`Set`, for every individual $P{:}A{\to}$`Prop`.
- `LIST_ind` can be applied to arbitrary A and an arbitrary P : $A{\to}$`Prop`

```
Fixpoint LENGTH [A:Set;l:(LIST A)] :nat :=
Cases l of | NIL => O
           | (CONS a k) => (S (LENGTH A k))
end.

Eval Compute in
   (LENGTH ? (CONS ? O (CONS ? O (CONS ? O (NIL ?))))).
>>    = (3) : nat

Syntactic Definition Nil := (NIL ?).
Syntactic Definition Cons := (CONS ?).
Syntactic Definition Length := (LENGTH ?).

Eval Compute in (Length (Cons O (Cons O (Cons  O Nil)))).
>>    = (3) : nat
```

```
Fixpoint SUMLIST [l:(LIST nat)]: nat :=
Cases l of | NIL => O
           | (CONS n k) => (plus n (SUMLIST k))
end.

Eval Compute in
   (SUMLIST (Cons (2) (Cons (2) (Cons  (1) Nil)))).

>>    = (5) : nat
```