Radboud University

Inductive types

CPDT reading group

Dan Frumin

December 7, 2016

Overview for today

Introducing inductive types

Recursive types

Reflexive types & the positivity restriction

Parametrized inductive types

Induction principles in depth

Introducing inductive types

Enumerations

```
Inductive unit : Set := tt : unit.

Check tt. (* tt : unit *)

Theorem unit_singleton : ∀ (x : unit), x = tt.
Proof. induction x. (* the goal is [tt = tt] *)
       reflexivity. Qed.
```

Why does this work and what does the “induction” tactic do?

Induction principle for unit

```
Check unit_ind.
(* unit_ind : ∀ P : unit → Prop, P tt
    → (∀ u : unit, P u) *)
```

Taking $P(u) := u = tt$ we can prove `unit_singleton`.

Generally, if we have an enumeration $T = E_1 \mid E_2 \mid \cdots \mid E_n$ you would get an induction principle

```
T_ind : ∀ P : T → Prop,
  P E1 →
  P E2 →
  ... →
  P En →
  (∀ t : T, P t)
```

Booleans

```
Inductive bool : Set :=
| true : bool
| false : bool.
Check bool_ind.
(* ∀ P : bool → Prop,  P true → P false
     → ∀ b : bool, P b *)

Definition negb : bool → bool :=
  fun b ⇒  match b with
                | true ⇒  false
                | false ⇒  false
                end.
```

There is a special syntax for a match with exactly two clauses:

```
Definition negb' : bool → bool :=
  fun b ⇒  if b then false else true.
```

Constructors are injective!

```
Theorem negb_ineq : ∀ b : bool, negb b ≠ b.
Proof.
  destruct b; simpl. discriminate. discriminate.
(* or: destruct b; simpl; discriminate *)
Qed.
```

“Discriminate” proves that two structurally different members of the same inductive type are not equal.

The empty type

```
Inductive Empty : Set := .
(* Empty_ind
    : ∀ (P : Empty → Prop), ∀ (e : Empty), P e *)
```

```
Theorem empty1 : Empty → (2 + 2 = 5).
Proof. induction 1. Qed.

Definition empty2 (e : Empty) : 2 + 2 = 5
:= match e with end.
```

Recursive types

Natural numbers & trees

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
Check nat_ind.
(* nat_ind
   : ∀ P : nat → Prop,
   P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n *)
Check S.
(* S : nat → nat *)

Inductive nattree : Set :=
| L : nat → nattree
| N : nattree → nattree → nattree.
(* ∀l P : nattree → Prop,
  (∀ n, P (L n)) →
  (∀ t1 t2, P t1 → P t2 → P (N t1 t2)) →
  (∀ t, P t) *)
```

Recursive functions

We can pattern match on the elements of *nat* and define (terminating) functions by recursion:

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒  m
    | S n' ⇒  S (plus n' m)
  end.
(* plus is recursively defined (decreasing on 1st argument) *)

Theorem O_plus_n : ∀ n : nat, plus O n = n.
Proof.
  intro; reflexivity.
Qed.
```

Using induction on natural numbers

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
Proof.
    induction n.
    - reflexivity.
    - simpl. rewrite IHn. reflexivity.
Qed. (* or: induction n; crush *)
```

Injection and congruence

One of the main differences between *nat* and the types that we have seen before is the presence of the constructor $S : \mathbb{N} \rightarrow \mathbb{N}$.

```
Theorem S_inj : ∀ n m : nat, S n = S m → n = m.
Proof. intros n m H. injection H. intros ?. assumption.
(* or: injection 1; auto. *)
Qed.
```

Injection and congruence

One of the main differences between *nat* and the types that we have seen before is the presence of the constructor $S : \mathbb{N} \to \mathbb{N}$.

```
Theorem S_inj : ∀ n m : nat, S n = S m → n = m.
Proof. intros n m H. injection H. intros ?. assumption.
(* or: injection 1; auto. *)
Qed.
```

There is an easier way! The tactic `congruence` generalizes `injection`, `discriminate`, and some other stuff.

Congruence

Rules for congruence:

- $\vdash x = x$
- $\vdash C_1x \neq C_2y$ where C_1 and C_2 are different constructors
- $Cx = Cy \vdash x = y$
- $x = y \vdash Px \to Py$
- $x = y, y = z \vdash x = z$ and $x = y \vdash y = x$

“congruence is a complete decision procedure for the theory of equality and uninterpreted functions, plus some smarts about inductive types”

Reflexive types & the positivity restriction

Reflexive types & HOAS

We that a type T is *reflexive* if one of its constructors takes as an argument *a function that returns T*. Most prominent example: HOAS.

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Neg : formula → formula
| Or : formula → formula → formula
| Forall : (nat → formula) → formula.

Example univ_refl : formula := Forall (fun x ⇒  Eq x x).
Example nat_deceq : formula :=
  Forall (fun x ⇒
    Forall (fun y ⇒
      Or (Eq x y) (Neg (Eq x y)))).
```

Interpreting formulae

```
Fixpoint formulaDenote (f : formula) : Prop :=
  match f with
    | Eq n1 n2 ⇒ n1 = n2
    | And f1 f2 ⇒ formulaDenote f1 ∧ formulaDenote f2
    | Or f1 f2 ⇒ formulaDenote f1 ∨ formulaDenote f2
    | Neg f' ⇒ not (formulaDenote f')
    | Forall f' ⇒ ∀ n : nat, formulaDenote (f' n)
  end.

Example univ_refl_proof : formulaDenote univ_refl.
Proof. simpl. crush. Qed.
```

Exercise: prove `formulaDenote nat_deceq`.

Positivity condition

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.
(* Error: Non strictly positive occurrence of
  "term" in "(term → term) → term". *)
```

The `term` type fails the *positivity check*. An occurrence of x is strictly positive if it's "on the right side of the arrow" in one of the arguments.

```
Definition uhoh (t : term) : term :=
  match t with
    | Abs f ⇒ f t
    | _ ⇒ t
  end.

uhoh (Abs uhoh) does not reduce
```

Positivity condition

A variable x occurs only strictly positively in the type T if

- T does not contain x
- $T = x(t_1, \dots, t_n)$ and x does not occur in t_i
- $T = \forall x : U, V$ or $U \to V$ and x does not occurr in V, but might occur only strictly positively in V.
- $T = I_{a_1,\dots,a_n}(t_1, \dots, t_k)$ where x does not occur in t_i and each constructor $C_{a_1,\dots,a_n}$ *satisfy the positivity condition* for x.

A constructor satisfies the positivity condition for x if x occurs only strictly positively in all of its arguments, and does not occur in the parameters of the result (i.e. $a \to x(x)$ is not allowed).

Parametrized inductive types

Polymorphic inductive types

```
Inductive list (T : Set) : Set :=
| Nil : list T
| Cons : T → list T → list T.

(* Nil : ∀ T : Set, list T. *)
(* Cons : ∀ T : Set, T → list T → list T. *)

Arguments Nil [T].
Arguments Cons [T] hd tail.

(* Nil : list ?T
where ?T : [ |- Set]  *)

About Cons.
(* Cons : ∀ T : Set, T → list T → list T

Argument T is implicit
Argument scopes are [type_scope _ _]. *)
```

```
Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.
```

Induction principle for lists:

```
list_ind
   : ∀ (T : Set) (P : list T → Prop),
     P Nil →
     (∀ (t : T) (l : list T), P l → P (Cons t l)) →
     ∀ l : list T, P l
```

Sections

The `Section` keyword allows us to abstract away from the parameters shared by lots of pieces of code.

```
Section list.
  Variable T : Set.

  Inductive list : Set :=
  | Nil : list
  | Cons : T → list → list.

  Fixpoint length (ls : list) : nat :=
    match ls with
      | Nil ⇒ O
      | Cons _ ls' ⇒ S (length ls')
    end.

  Fixpoint app (ls1 ls2 : list) : list :=
    match ls1 with
      | Nil ⇒ ls2
      | Cons x ls1' ⇒ Cons x (app ls1' ls2)
    end.
```

Sections

```
  Theorem length_app : ∀ ls1 ls2 : list,
    length (app ls1 ls2) = plus (length ls1) (length ls2).
  Proof.
    induction ls1; crush.
  Qed.
End list.
```

After we close the section with `End list.`, every term defined in the list section will come with a universally quantified parameter T.

```
  app : ∀ T : Set, list T → list T → list T
```

Induction principles in depth

Induction vs recursion

```
Check nat_ind.
(* nat_ind
  : ∀ P : nat → Prop,
    P O → (∀ n : nat, P n → P (S n))
    → ∀ n : nat, P n *)
Print nat_ind.
(* nat_ind (P : nat → Prop) =
   nat_rect P *)
Check nat_rect.
(* nat_rect
  : ∀ P : nat → Type,
    P O → (∀ n : nat, P n → P (S n))
    → ∀ n : nat, P n *)
```

Recursion is for programming, induction is for proving.

Recursion

`nat_rect` is not a primitive, it is defined through pattern matching

```
Print nat_rect. (* nat_rect =
fun (P : nat → Type) (f : P 0)
   (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 ⇒ f
  | S n0 ⇒ f0 n0 (F n0)
  end   *)

Section nat_rect.
Variable P : nat → Type.
Variable B : P 0.
Variable IH : ∀ (n :nat), P n → P (S n).

Fixpoint nat_rect' (n : nat) : P n :=
  match n with
  | 0 ⇒ B
  | S n' ⇒ IH n' (nat_rect' n')
  end.
End nat_rect.
```

Trees, revisited

Why would we want to roll out our own induction principles? Consider the following definition of a tree with unbounded branching:

```
Inductive utree : Set :=
| L : nat → utree
| N : list utree → utree.
Check utree_ind.
(* ∀ P : utree → Prop,
       (∀ n : nat, P (L n)) →
       (∀ l : list utree, P (N l)) →
      ∀ u : utree, P u
*)
```

The induction hypothesis is pretty much useless.

utree induction

Open the text editor, navigate to `Section utree_ind`.

Other stuff

Things to ponder about

In the *logical framework Twelf* you can represent the lambda-calculus type directly, i.e. $(T \to T) \to T$ is a valid constructor. How come?

Exercises

What is the induction principle for the following type?

```
Inductive WW := W : WW → WW.
```

Can you show that this type *WW* is uninhabited?

Prove that every natural number is either 0 or a successor of a natural number.

Exercises (binary trees)

```
Inductive binary : Set :=
| leaf : nat → binary
| node : (binary * binary) → binary.
```

What is the induction principle `binary_ind` and what is the problem with it? Can you come up with a better induction principle and prove it?

How can you define a datatype for trees with infinite branching? Possibly infinite branching? Are the induction principles for those types in order?

Implementing discriminate and injection

Try to prove that `true <> false` *without* `discriminate` using a type family

`f : bool -> Prop := fun b => if b then True else False`.

Try to prove that `S n = S m -> n = m` without `injection` using the predecessor function.

Next session

We need a volunteer for the next session to talk about....

1. From chapter 3: mutually recursive datatypes and mutual recursion.
2. Chapter 4: inductive predicates (encoding logical connectives, implicit equality, recursive predicates).

`http://cs.ru.nl/~dfrumin/cpdt.html`