

logical verification lecture 4

2011 04 12

inductive definitions

overview

- logic:
minimal, plus falsum, intuitionistic, classical
- lambda calculus:
simply typed
- Curry-Howard-de Bruijn isomorphism:
 $ML \sim \lambda \rightarrow$
- inductive definitions

formulas as types and proofs as terms

proof : formula

~

term : type

formulas as types and proofs as terms

$$\frac{\frac{[A^x]}{B \rightarrow A} \text{I}[y] \rightarrow}{A \rightarrow B \rightarrow A} \text{I}[x] \rightarrow}{:} \quad A \rightarrow B \rightarrow A$$

~

$$\lambda x:A. \lambda y:B. x \quad : \quad A \rightarrow B \rightarrow A$$

towards program extraction

every finite list of natural numbers can be sorted

$\forall l : \text{natlist} \quad \exists k : \text{natlist} \quad \text{Sorted}(k) \wedge \text{Permutation}(k, l)$

in order to express this we need:

- universal and existential quantification (\forall, \exists)
- data-types (via inductive definitions) (natlist)
- predicates (via inductive definitions) ($\text{Sorted}, \text{Permutation}$)

we will find:

$P : \forall l : \text{natlist} \quad \exists k : \text{natlist} \quad \text{Sorted}(k) \wedge \text{Permutation}(k, l)$

which is a λ -term (executable program) representing a proof

Check

in Coq everything is typed

use `Check` to see the type

examples: types

initial declaration: Parameter A B : Prop.

- A : Prop
- A -> B : Prop
- fun (x:A) (y:B) => x : A -> B -> A : Prop
- Prop : Type

examples: types

```
Inductive nat : Set :=  
  0 : nat  
| S : nat -> nat .
```

- `nat : Set`
- `nat -> nat : Set`
- `0 : nat : Set`
- `S: nat -> nat : Set`
- `Set : Type`

universes of Coq

- Prop
intuitively for propositions or formulas
Prop : Type
- Set
intuitively for data-types
Set : Type
- Type
in fact an infinite hierarchy

universes of Coq: example

term : type : kind

(S 0) : nat : Set : Type

fun x:A => x : A->A : Prop : Type

inductive types in each universe

- inductive data-types:

`Inductive ... : Set :=`

- inductive predicates:

`Inductive ... : Prop :=`

- `Inductive ... : Type :=`

inductive type: example

```
Inductive nat : Set :=  
  0 : nat  
| S : nat -> nat .
```

inductive definitions

- intuitively: minimal set closed under constructors
- functions using pattern matching and recursion
- proofs by cases and using induction

proving properties: example

Lemma plus_n_0 : forall n : nat, plus n 0 = n.

Proof.

induction n.

(* alternative: intro n. elim n *)

(* case n = 0 *)

simpl.

reflexivity.

(* case n > 0 *)

simpl.

rewrite IHn.

reflexivity.

nat and its induction principle

```
Inductive nat : Set :=  
  0 : nat  
| S : nat -> nat .
```

```
nat_ind :  
  forall P : nat -> Prop,  
  P 0 ->  
  (forall n : nat, P n -> P (S n)) ->  
  forall n : nat, P n
```

induction on natural numbers

```
nat_ind :  
  forall P : nat -> Prop,  
  P 0 ->  
  (forall n : nat, P n -> P (S n)) ->  
  forall n : nat, P n
```

Goal:

```
forall n : nat, n + 0 = n
```

we can do:

```
apply nat_ind .
```

induction on natural numbers

the tactic `induction` roughly does apply `nat_ind`

induction principles

every inductive data-type comes with an induction principle

even

```
Inductive even : nat -> Prop :=  
| even0 : even 0  
| evenSS : forall n:nat ,  
           even n -> even (S (S n)) .
```

even: examples

```

                                even0 : even 0 : Prop
                                even 1 : Prop
        evenSS 0 even0 : even 2 : Prop
                                even 3 : Prop
evenSS 2 (evenSS 0 even0) : even 4 : Prop
```

even and odd

```
Inductive ev : nat -> Prop :=
```

```
| ev0 : ev 0
```

```
| evS : forall n:nat , odd n -> ev (S n)
```

```
with odd : nat -> Prop :=
```

```
| oddS : forall n:nat , ev n -> odd (S n) .
```

less than or equal

```
Inductive le (n:nat) : nat -> Prop :=  
| le_n : le n n  
| le_S : forall m:nat , le n m -> le n (S m) .
```

less than or equal: examples

```
le_n 0 : le 0 0 : Prop
le_n 7 : le 7 7 : Prop
le_S 0 0 (le_n 0) : le 0 1 : Prop
le_S 0 1 (le_S 0 0 (le_n 0)) : le 0 2 : Prop
```

inductive definition of the type of lists of natural numbers

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

inductive predicate for sorted

```
Inductive sorted : natlist -> Prop :=  
| sorted0 : sorted nil  
| sorted1 : forall n:nat , sorted (cons n nil)  
| sorted2 : forall n h:nat , forall t:natlist ,  
    le n h ->  
    sorted (cons h t) ->  
    sorted (cons n (cons h t)) .
```

inductive predicate for permutation

```
Inductive permutation : natlist->natlist->Prop :=  
  | permutation_nil : permutation nil nil  
  | permutation_cons :  
    forall (n : nat) (l l' l'' : natlist),  
    permutation l l' -> inserted n l' l'' ->  
      permutation (cons n l) l''.
```

how to define inductive predicates

- constructors are axioms and should be intuitively true
- constructors define mutually exclusive cases
- test positive and negative cases