

logical verification lecture 3  
 2011 04 05  
 dynamics  
 inductive definitions

## overview: propositional logic

- minimal logic (ML)  
 $(((((A \rightarrow B) \rightarrow A) \rightarrow A) \rightarrow B) \rightarrow B)$
- ML +  $\perp$   
 $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$
- intuitionistic logic  
 $A \vee B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$
- classical logic  
 $A \vee \neg A$
- we will add (later):  
 quantification over terms and over predicates

## overview: simply typed lambda calculus

- variable rule:  
 $\Gamma \vdash x : A \quad \text{if } x : A \in \Gamma$
- abstraction rule:  

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A. M) : A \rightarrow B}$$
- application rule:  

$$\frac{\Gamma \vdash F : A \rightarrow B \quad \Gamma \vdash M : A}{\Gamma \vdash (F M) : B}$$
- we will add:  
 dynamics and inductive definitions

## overview: Curry-Howard-De Bruijn isomorphism

prop1	~	$\lambda \rightarrow$
formula	~	type
propositional variable	~	type variable
connective $\rightarrow$	~	type constructor $\rightarrow$
proof	~	term
assumption	~	term variable
implication introduction	~	abstraction
implication elimination	~	application

we will add:  
 dynamics

## preview

- more expressive logics:  
second order, predicates
- more expressive lambda calculi: polymorphism, dependent types, inductive definitions
- more Coq tactics

## Curry-Howard-De Bruijn isomorphism

type checking:

is the typing derivation  $\Gamma \vdash P : A$  correct?

corresponds to

proof checking

is the proof  $P$  of the formula  $A$  using assumptions  $\Gamma$  correct?

and is decidable for  $\lambda^{\rightarrow}$  and ML

## Curry-Howard-De Bruijn isomorphism

inhabitation

do we have a closed term  $P$  of type  $A$ ?

corresponds to

provability

do we have a proof  $P$  of the formula  $A$ ?

decidable for  $\lambda^{\rightarrow}$  and ML

(but for instance not for first-order predicate logic)

## formulas as types and proofs as terms

proof : formula

$\sim$

term : type

## formulas as types and proofs as terms

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

~

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

## term normalization: beta-reduction

a  $\lambda$ -term may be reduced or rewritten or evaluated

## 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 (natlist)
- predicates (Sorted, 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  $\lambda$ -term (executable program) representing a proof

## beta-reduction: definitions

$\beta$ -reduction rule:

$(\lambda x : A. M) N \rightarrow_{\beta} M[x := N]$

$\beta$ -reduction step:

application of the rule in a context (a bigger term)

$\beta$ -reduction:

a sequence of  $\beta$ -reduction steps

## beta-reduction: examples

- $(\lambda f : \text{nat} \rightarrow \text{nat}. \lambda x : \text{nat}. f\ x)\ \text{even}\ 3 \rightarrow_{\beta} ?$
- $(\lambda f : \text{nat} \rightarrow \text{nat}. f\ 2)\ \lambda x : \text{nat}. x \rightarrow_{\beta} ?$
- $(\lambda x : \text{nat}. f\ x\ x)\ 2 \rightarrow_{\beta} ?$
- $(\lambda x : \text{nat}. 2)\ 3 \rightarrow_{\beta} ?$

## beta-reduction: subject reduction

types are preserved under computation

if  $\Gamma \vdash M : A$  and  $M \rightarrow_{\beta} M'$  then  $\Gamma \vdash M' : A$

## beta-reduction: normal form

$\beta$ -redex:

sub-term of the form  $(\lambda x : A. M)\ N$

normal form:

term without a  $\beta$ -redex

## beta-reduction: unique normal forms

result of a computation is unique

via confluence:

if  $M \twoheadrightarrow_{\beta} N$  and  $M \twoheadrightarrow_{\beta} P$   
then there is  $Q$  such that:  
 $N \twoheadrightarrow_{\beta} Q$  and  $P \twoheadrightarrow_{\beta} Q$

beta-reduction: termination

and for proofs?

all computations eventually end in simply typed  $\lambda$ -calculus

(in untyped  $\lambda$ -calculus there are infinite computations)

a  $\lambda$ -term represents a proof

are proofs also evaluated?

yes: proof normalization

proof normalization

proof normalization: detour

we restrict attention to ML (only implication)

introduction immediately followed by an elimination

$$\frac{\frac{\frac{\vdots}{C}}{A \rightarrow C} \text{I}[x] \rightarrow}{C} \frac{\vdots}{A} E \rightarrow$$

## detour: example

$$\frac{\frac{\frac{[A^y]}{B \rightarrow A} I[z] \rightarrow}{A \rightarrow B \rightarrow A} I[y] \rightarrow}{\frac{B \rightarrow A}{A \rightarrow B \rightarrow A} I[x] \rightarrow} E \rightarrow [A^x]$$

## detour elimination: example

$$\frac{\frac{\frac{[A^y]}{B \rightarrow A} I[z] \rightarrow}{A \rightarrow B \rightarrow A} I[y] \rightarrow}{\frac{B \rightarrow A}{A \rightarrow B \rightarrow A} I[x] \rightarrow} E \rightarrow [A^x]$$

reduces to

$$\frac{[A^x]}{A \rightarrow B \rightarrow A} I[x] \rightarrow$$

## proof normalization: detour elimination

$$\frac{\frac{\frac{\vdots}{C}}{A \rightarrow C} I[x] \rightarrow}{C} \frac{\frac{\vdots}{A} E \rightarrow}{E \rightarrow}$$

is replaced by

$$\frac{\vdots}{C}$$

where every occurrence of the assumption  $A^x$  is replaced by the proof

$$\frac{\vdots}{A}$$

## proof normalization: normal proof

proof without a detour

what about the isomorphism?

Curry-Howard-De Bruijn isomorphism

is term normalization related to proof normalization?

yes: this is the dynamic part of the Curry-Howard isomorphism

proof normalization  $\sim$   $\beta$ -reduction

detour  $\sim$  redex  
normalization step  $\sim$  reduction step  
normal proof  $\sim$  normal form

we add inductive definitions

- to express data-types such as natural numbers
- to express logical operations such as conjunction
- to express predicates such as even

examples of inductive definitions

- booleans
- natural numbers
- finite lists
- binary trees
- pairs
- False, True
- conjunction, disjunction

## inductive type: example

```
Inductive day : Set :=  
  | monday : day  
  | tuesday : day  
  | wednesday : day  
  | thursday : day  
  | friday : day  
  | saturday : day  
  | sunday : day.
```

## evaluating

```
Eval simpl in (next_day friday).  
Eval compute in (next_day friday).
```

## pattern matching: example

```
Definition next_day (d:day) : day :=  
  match d with  
  | monday => tuesday  
  | tuesday => wednesday  
  | wednesday => thursday  
  | thursday => friday  
  | friday => saturday  
  | saturday => sunday  
  | sunday => monday  
  end.
```

## proving

```
Lemma tomorrow : next_day friday = saturday.  
  
Proof.  
simpl.  
reflexivity.  
Qed.
```

## programming by proving

later we will see how to extract a program from a proof

## inductive type: example

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

## inductive types: ingredients

- a new type  
`nat:Set`
- constructor functions  
`0:nat`  
`S : nat -> nat`
- minimal set closed under applying the constructors

## inductive types: how to define functions?

- pattern matching
- recursion
- evaluation by  $\iota$ -reduction

## pattern matching and recursion: example

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | (S p) => S (plus p m)
  end .
```

## evaluating

```
Eval compute in (plus (S 0) (S 0)) .
```

## inductive types: how to prove properties?

- proof by cases
- proof by 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.
```

## natural numbers: 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
```

## natural numbers: induction

## natural numbers: induction

```
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 .
```

the tactic `induction` roughly does `apply nat_ind`