Computational Logic

Interactive Theorem Proving

Week 10

Cezary Kaliszyk

May 19, 2015

Summary

So far

- Proof Assistants: HOL Light, Mizar, Isabelle
- Calculi: $\lambda_{\rightarrow}$, λ_P, λ_2
- Curry-Howard

Today

- Inductive types in $\lambda_{\star}$
- λ_{CoC}

Booleans (in Coq)

Inductive type definition

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

(Coq produces `bool_ind`, ...)

Recursive definition

```
Definition neg (b : bool) : bool :=
  match b with
    true => false
  | false => true
  end.
```

First properties

Proof by cases

```
  Lemma negneg : forall b : bool, neg (neg b) = b
intro b.
elim b.
simpl.
reflexivity.
simpl.
reflexivity.
Qed.
```

Induction principle

```
Check bool_ind.
bool_ind :
  forall P : bool -> Prop,
    P true -> P false -> forall b : bool, P b
```

$$\forall P \in (\mathsf{bool} \to \mathsf{Prop}).\ P(\mathsf{true}) \Rightarrow P(\mathsf{false}) \Rightarrow\ \forall b \in \mathsf{bool}.\, P(b)$$

Natural numbers

Inductive Definition

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

Recursive function

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    O => m
  | S p => S (plus p m)
  end.
```

ι-reduction

```
plus (S O) (S O) ↠ S (plus O (S O)) ↠ S (S O)
```

Proof by Induction

Lemma plus_rid: forall n : nat, plus n O = n.

```
induction n.
simpl.
reflexivity.
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Induction Principle

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

$$\forall P.\ P(0) \Rightarrow (\forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}.\ P(n)$$

Lists

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

Fixpoint append (l k : natlist) : natlist :=
  match l with
    nil => k
  | cons n l' => cons n (append l' k)
  end.
```

$\iota^\star$-reduction

```
Eval compute in (append (cons 0 nil) (cons (S 0) nil)).
```

Inductive Predicates

Truth as a type

```
Inductive True : Prop :=
  I : True.
```

Falsity as a type

```
Inductive False : Prop :=
  .
```

Generated induction principle

```
  False_ind :
    forall P : Prop, False -> P
```

Universes: Prop versus Set

Types

```
fun x : A => ...
forall x : A, ...
```

```
A : Prop
A : Set
A : Type
```

```
S O  :  nat  :  Set
                :
                Type  :  Type  :  ...
                :
fun x:A => x  :  A -> A  :  Prop
```

Universes: Prop versus Set

Types

```
fun x : A => ...
forall x : A, ...
A : Prop
A : Set
A : Type
```

```
S O  :  nat  :  Set
                :
                Type 0  :  Type 1  :  ...
                :
fun x:A => x  :  A -> A  :  Prop
```

Quiz

Inductive types?

Universes: Prop versus Set

Types

```
fun x : A => ...
forall x : A, ...
A : Prop
A : Set
A : Type
```

```
S O  :  nat  :  Set
                :
                Type 0  :  Type 1  :  ...
                :
fun x:A => x  :  A -> A  :  Prop
```

Quiz

Inductive types? `True` and `bool`
Curry-Howard?

Universes: Prop versus Set

Types

```
fun x : A => ...
forall x : A, ...
A : Prop
A : Set
A : Type
```

```
S O  :  nat  :  Set
                :
                Type 0  :  Type 1  :  ...
                :
fun x:A => x  :  A -> A  :  Prop
```

Quiz

Inductive types? `True` and `bool`
Curry-Howard? Only for `True` and `Prop`

Datatypes in λ-calculus

Church numerals in untyped λ-calculus

$$
\begin{aligned}
0 &= \lambda f.\, \lambda x.\, x \\
1 &= \lambda f.\, \lambda x.\, f\, x \\
2 &= \lambda f.\, \lambda x.\, f\, (f\, x) \\
&\vdots \\
S &= \lambda n.\, \lambda f.\, \lambda x.\, f\, (n\, f\, x)
\end{aligned}
$$

Church numerals can be typed

type A

$$
\begin{aligned}
0 &= \lambda f : A \to A.\, \lambda x : A.\, x \quad : \quad (A \to A) \to (A \to A) \\
S &= \lambda n : (A \to A) \to (A \to A).\, \lambda f : A \to A.\, \lambda x : A.\, f\, (n\, f\, x)
\end{aligned}
$$

Why included if definable?

Efficiency, Separation of reductions.

Dependently-typed polymorphic lists

```
Inductive t A : nat -> Type :=
  |nil : t A 0
  |cons : forall (h:A) (n:nat), t A n -> t A (S n).

Check (t nat 0).
Check (cons _ 3 _ (nil _) ).

Local Notation "[]" := (nil _).
Local Notation "h :: t" := (cons _ h _ t) (at level 60, right associativity).

Fixpoint const A (a:A) (n:nat) :=
  match n return t A n with
  | 0 => nil A
  | S n => a :: (const a n)
  end.

Fixpoint of_list A (l : list A) : t A (length l) :=
match l as l' return t A (length l') with
  |Datatypes.nil => []
  |(h :: tail)%list => (h :: (of_list tail))
end.
```

Summary

Today

- Calculus of Constructions
- Introduction to Coq

Next time

- Proofs as programs
- Code extraction, code generation