overview

impredicative definitions of data-types

impredicativity

the ‘meaning’ of $\forall a.\, B$

depends on the meaning of all formulas $B[a := A]$

(this includes $B[a := \forall a.\, B]$)

Russell's paradox

naive set theory: $\mathcal{P}(\mathtt{Set}) \subseteq \mathtt{Set}$

Russell's paradox:

$$\{x \mid x \notin x\} \in \{x \mid x \notin x\}$$
$$\Leftrightarrow$$
$$\{x \mid x \notin x\} \notin \{x \mid x \notin x\}$$

Cantor: powerset bigger than set itself

lambda2 is impredicative

$\mathsf{bool} : \star \vdash \star \to \mathsf{bool} : \star$

corresponds to

$\mathcal{P}(\mathtt{Set}) \subseteq \mathtt{Set}$

Coq is mixed

Prop is impredicative

```
forall x : Prop, x          : Prop
forall x : Prop, x -> True : Prop
```

Set is not impredicative

```
forall x : Set, x          : Type
forall x : Set, x -> bool : Type
```

use of polymorphic types

alternative to inductive definitions

we can define logical connectives (in Prop) in prop2

also: we can define various datatypes (in Set) in lambda2

booleans in lambda2

- $\mathsf{B} = \Pi a{:}* \,.\, a \to a \to a$
- $\mathsf{T} = \lambda a : * \,.\, \lambda x : a.\, \lambda y : a.\, x$
- $\mathsf{F} = \lambda a : * \,.\, \lambda x : a.\, \lambda y : a.\, y$
- $\mathsf{not} = \lambda b : \mathsf{B}.\, \lambda a : * \,.\, \lambda u : a.\, \lambda v : a.\, (b\ a\ v\ u)$
- $\mathsf{and} = \lambda b : \mathsf{B}.\, \lambda b' : \mathsf{B}.\, \lambda a : * \,.\, \lambda u : a.\, \lambda v : a.\, (b\ a\ (b'\ a\ u\ v)\ v)$
- $\mathsf{or} = \lambda b : \mathsf{B}.\, \lambda b' : \mathsf{B}.\, \lambda a : * \,.\, \lambda u : a.\, \lambda v : a.\, (b\ a\ u\ (b'\ a\ u\ v))$

natural numbers in lambda2

- $\mathsf{N} = \Pi a{:}* \,.\, a \to (a \to a) \to a$
- $\mathsf{n} = \lambda a : * \,.\, \lambda o : a.\, \lambda s : a \to a.\, s^n o$
- $\mathsf{successor} = \lambda n : \mathsf{N}.\, \lambda a : * \,.\, \lambda o : a.\, \lambda s : a \to a.\, s\,(n\,a\,o\,s)$
- $\mathsf{successorb} = \lambda n : \mathsf{N}.\, \lambda a : * \,.\, \lambda o : a.\, \lambda s : a \to a.\, (n\,a\,(s\,o)\,s)$

adding inductive types

type theory = typed λ-calculus + inductive types

Curry-Howard-De Bruijn isomorphism

logic	$\sim$	type theory
formula	$\sim$	type
proof	$\sim$	term
detour elimination	$\sim$	β-reduction
minimal prop1	$\sim$	simply typed λ-calculus
prop1	$\sim$	simply typed λ-calculus + inductive types
classical prop1	$\sim$	add exceptions

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

overview

inductive definitions: datatypes

further reading

examples of inductive types

booleans

natural numbers

lists (of natural numbers)

binary trees

logical operations

inductive datatype: nat

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

some types:

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

inductive types

types

intuitively: minimal set closed under constructors

recursive functions

definition using pattern matching

evaluation using ι-reduction

proof by cases

induction principle

nat and natlist: definitions

a nat is either zero or successor of some nat

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

a natlist is either nil or cons of some nat and some natlist

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

nat and natlist: functions

pattern matching considers two cases for a nat

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

pattern matching considers two cases for a natlist

```
Fixpoint append (k l : natlist) {struct k} : natlist :=
  match k with
    nil => l
  | (cons h t) => cons h (append t l)
  end .
```

computing in Coq

ι-reduction

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

inductive data-types: induction principles

every inductive data-type comes with an induction principle

nat and natlist: induction principles

induction on nat

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

induction on natlist

```
natlist_ind
     : forall P : natlist -> Prop,
       P nil ->
       (forall (n : nat) (n0 : natlist), P n0 -> P (cons n
       forall n : natlist, P n
```

nat and natlist: tactic induction

for some n in nat

`induction n` is roughly `apply nat_ind`

for some l in natlist

`induction l` is roughly `apply natlist_ind`

nat and natlist: proofs

NB recursion of plus in the first argument

```
Lemma plus_n_O : forall n : nat, plus n O = n.
```

NB recursion of append in the first argument

```
Lemma append_l_nil : forall l : natlist , append l nil = l
```

NB: sometimes the proof makes you reconsider your definitions

induction: example on nat

```
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 .
```

proving properties: example with plus

```
Lemma plus_n_O : forall n : nat, plus n O = n.
Proof.

induction n.
(* alternative: intro n. elim n *)

(* case n = O *)
simpl.
reflexivity.

(* case n > O *)
simpl.
rewrite IHn.
reflexivity.
```

tactics we use a lot

```
induction n

simpl

reflexivity

rewrite IHn

elim n
```

tactic discriminate

if a hypothesis has the form

```
H : O = S O
```

use the tactic

```
discriminate H.
```

in general: for hypotheses $s = t$ where s and t start with different constructors

tactic injection

if a hypothesis has the form

```
H : S n = S m
```

use the tactic

```
injection H.
```

in general: for hypothesis $s = t$ where s and t start with the same constructors

dependent types in programming: further reading

- Epigram (Conor McBride, James McKinna)
- Dependent ML (Frank Pfenning and Hongwei Xi)
- Dependent types in Haskell

schema

inductive definitions: predicates

inductive predicates: logical connectives

overview

inductive definitions: predicates

inductive predicates: logical connectives

inductive type

a new type

constructor functions

pattern matching and ι-reduction

induction principle

inductive types in each universe

- inductive data-types:
  `Inductive ... : Set :=`
- inductive predicates:
  `Inductive ... : Prop :=`
- `Inductive ... : Type :=`

universes of Coq: example

term	:	type	:	kind		
`(S O)`	:	`nat`	:	`Set`	:	`Type`
`fun x:A => x`	:	`A->A`	:	`Prop`	:	`Type`

program extraction rough idea

an intuitionistic (constructive) proof

corresponds to an executable algorithm

constructive functional programming

- program specification
- constructive proof of existence
- automatically generated functional program

program specification: example

the correctness proof of the specification

$$\forall l : \mathit{natlist}.\, \exists l' : \mathit{natlist}.\, \mathit{permutation}(l, l') \wedge \mathit{sorted}(l')$$

yields a program (function) from natlist to natlist

program specification: general pattern

$$\forall x : A.\, P(x) \to \exists y : B.\, Q(x, y)$$

A	input type
B	output type
$P(x)$	precondition
$Q(x, y)$	input/output behaviour

the correctness proof yields a program from A to B

program extraction in Coq

Coq proof in type theory gives

functional program in ML or Haskell or Scheme

program extraction in Coq

is almost the identity function but

- other typing system
- information from Prop is erased

existential quantification in Prop

inductive type:

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P
```

syntax:

```
exists x : A , P x.
```

existential quantification in Set

inductive type:

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
    exist : forall x : A, P x -> sig P
```

syntax:

```
{x:A | P x}
```

for program extraction

use existential quantification in Set

schema

program extraction

program extraction: examples

verified programs: alternative approach

examples lambda P

successor: existence proof and extracted program

specification:

```
Theorem successor:
  forall n:nat, {m:nat | m = S n} .
```

extracted program :

```
let successor n =
  S n
```

predecessor: existence proof and extracted program

specification:

```
Theorem predecessor :
  forall n:nat, ~(n=0) -> {m:nat | S m = n} .
```

extracted program :

```
let rec predecessor = function
  | 0 -> assert false (* absurd case *)
  | S n0 -> n0
```

insertion sort: existence proof

```
Theorem Sort :
  forall l : natlist,
  {l' : natlist | permutation l l' /\ sorted l'}.
```

insertion sort: predicate 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''.
```

insertion sort: predicate inserted

```
Inductive inserted (n:nat):
natlist->natlist->Prop :=
  | inserted_front :
      forall l : natlist, inserted n l (cons n l)
  | inserted_cons :
      forall (m : nat) (l l' : natlist),
        inserted n l l' ->
        inserted n (cons m l) (cons m l').
```

schema

program extraction

program extraction: examples

verified programs: alternative approach

examples lambda P

verified programs: two approaches

- correctness proofs
  from program to proof

- program extraction
  from proof to program

correctness proofs: Hoare logic

imperative program

$\rightsquigarrow$

annotated imperative program

$\rightsquigarrow$

proof obligations