Proof Assistants
Functional Programming 1

Bruno Barras and Christine Paulin-Mohring

INRIA, Université Paris-Sud 11

04/01/11
Overview

Inductive types

Introduction

Recursive functions
  Structural recursion
  Generalised recursion
  Well-founded recursion

Partial functions

Coinductive definitions
  Principles
  Streams
Recap on inductive types

- A general mechanism for
  - Algebraic datatypes: pairs, sums, lists, trees...
  - Propositional connectives: absurd, conjunction, disjunction
  - Predicate calculus: existential, equality
  - Inductive relations: inference rules

- Examples

```
Inductive False : Prop := .
Inductive True : Prop := I : True.
Inductive eq (A:Prop)(x:A) : A -> Prop := refl : eq x x.
Inductive empty : Type := .
Inductive unit : Type := I : True.
Inductive bool : Type := true : bool | false : bool.
```
Properties of inductive definitions

- Constructors: introduction rule
- Elimination: minimality principle
- Evaluation: the constructors are the values
- Two distinct constructors are they provably different?
Inductive types

Discriminable constructors

How to prove \( \text{true} \neq \text{false} \)?

Definition \( \text{b2p} \) (\( \text{b:bool} \)) : Type :=
\[
\text{match b with true => True | false => False end.}
\]

Definition \( \text{true_false} \) (\( \text{p:true=false} \)) : False :=
\[
\text{match p in _ = x return b2p x with}
\text{ | refl => I end.}
\]
Discriminable constructors

- \( C \ x_1 \ldots \ x_n <> D \ y_1 \ldots \ y_m \)
  for \( C \) and \( D \) two distinct constructors of an inductive type \( I \).
- Related to proof-irrelevance: two proofs of the same formula. \( \text{left } a \neq \text{right } a : A \lor A \) ?
- Case-analysis on \( I \) towards sort \( s \) with \( a \ b : A : s \) and \( a \neq b \).
  - \( s = \text{Type} : A = \text{Prop} a = \text{True} b = \text{False} \)
  - \( s = \text{Set} : \text{strong elimination towards Type} \)
  - \( s = \text{Prop} : \text{no elimination towards Set et Type (exception: singleton types, 0 or 1 constructor with arguments in Prop).} \)
Plan

Inductive types

Introduction

Recursive functions
  Structural recursion
  Generalised recursion
  Well-founded recursion

Partial functions

Coinductive definitions
  Principles
  Streams
Higher order logics

Languages for maths:

CIC

- Inductive types as base types:
- Higher-order functions seen as algorithms, which can be computed
  Reasoning modulo reduction rules.

≠ HOL or set theory

- Base typed $\iota$ and arrow types $\tau_1 \rightarrow \tau_2$
- Functions seen as relations.
  Need for a description operator: $\lambda x. \varepsilon y. R(x, y)$
  Computation not primitive.
Introduction

Programming language vs logical language

Partial terms: an expression might not denote a value:

► General fixpoint (let rec)

Definition negb (b:bool) : bool :=
  if b then false else true.
Fixpoint fixb : bool := negb fixb.

If fixb = negb fixb is provable, and also
forall b:bool, b=true \(\lor\) b=false

(dependent elimination)
Then true = false is provable.
Introduction

Programming language vs logical language

- Incomplete pattern-matching

Definition negb (b:bool) : bool :=
    match b with true => false end.

egb false is a normal closed term which is not a constructor.

Definition abs1 (b:bool) : False :=
    match b with end.

Definition abs2 (b:bool)
    : if b then bool else False
    := match b with true => true end.

abs1 true or abs2 false are closed terms of type False.
Plan

Inductive types

Introduction

Recursive functions
  Structural recursion
  Generalised recursion
  Well-founded recursion

Partial functions

Coinductive definitions
  Principles
  Streams
Recursive functions

Structural recursion

Fixpoint

\( \text{fix } f : B := t \)

- Typing: \( f : B \vdash t : B \)
- Structural recursion: there is \( n \) s.t.
  - \( B \equiv (x_1 : A_1) \ldots (x_n : A_n) B' \)
  - \( A_n \) is an inductive type
  - occurrences of \( f \) in \( t \) are of the form \( f u_1 \ldots u_n \) with \( u_n \) structurally smaller than \( x \)
    (essentially a variable of a pattern-matching on \( x \)).
- Reduction \( \text{fix } f : B := t \rightarrow t[f := \text{fix } f : B := t] \) only in expressions \( \text{fix } f : B := t y_1 \ldots y_n \) where \( y_n \) begins with a constructor.

Enough to encode structural recursion operator.
Fixpoint reduction

Preserve strong normalization: reduction of open terms without fixed strategy.

\[ \text{Fixpoint} \quad R \ (n : \text{nat}) \ : \ C := \]
\[ \text{match} \ n \ \text{with} \ O \ \Rightarrow \ x \ | \ S \ m \ \Rightarrow \ f \ m \ (R \ m) \ \text{end}. \]

- We want:
  - \( R \ 0 \ \rightarrow x \)
  - \( R \ (S \ m) \ \rightarrow f \ m \ (R \ m) \)

- Lose normalisation if:
  \[ R \ n \ \rightarrow \text{match} \ n \ \text{with} \ O \ \Rightarrow \ x \ | \ (S \ m) \ \Rightarrow \ f \ m \ (R \ m) \ \text{end}. \]
Defining general recursive functions

**fix** \( f \ x : C := t \) with non structural recursive calls.

**Fixpoint** \( \text{merge} \ (l1 \ l2:list) : list := \)

\[
\begin{align*}
\text{match} & \ (l1,l2) \ \text{with} \\
& \ \ [\] \ , \ _ \ & \Rightarrow \ l2 \\
& \ | \ _ \ , \ [] \ & \Rightarrow \ l1 \\
& \ | a::m1, b::m2 \ & \Rightarrow \ \text{if} \ a < b \ \text{then} \ a::\text{merge} \ m1 \ l2 \\
& \ & \quad \ \text{else} \ b::\text{merge} \ l1 \ m2 \ \text{end}. \\
\end{align*}
\]

- Extend the definition scheme (strong normalisation!).
- Or use encoding
  - Double recursion
  - Extra argument decreasing structurally
Recursive functions

Generalised recursion

Encoding recursion

Double recursion

**Fixpoint** `merge (l1 l2:list) : list :=`

```
match l1 with
  [] => l2
| a::m1 => (fix maux (m: list) : list :=
  match m with
    [] => l1
  | b::m2 => if a < b then a::merge m1 m
  else b::maux m2 end)
end.
```

**Reductions:**

- `merge [] l2 => l2`
- `merge (a :: m1) => fix maux m := match m with
  [] => a :: m1
  | b :: m2 => if a < b then a :: merge m1 m
  else b :: maux m2 end) l2 end.`
- `merge (a :: m1) [] => a :: m1`
- `merge (a :: m1) (b :: m2) => if a < b then
  a :: merge m1 (b :: m2)
else b :: merge (a :: m1) m2`
Encoding recursion

Extra argument: measure (length of \( l_1 \otimes l_2 \)).

```coq
Fixpoint maux (l1 l2:list) (n:nat) : list :=
  match (l1,l2) with
  | [], _, n => l2                |_ _ , [], n => l1
  | _, _, 0 => []                |_ _ , _, 0 => [] (* absurd case if |l1@l2| <= n *)
  | a::m1,b::m2,S n => if a < b then a::maux m1 l2 n
                        else b::maux l1 m2 n end.
```

**Definition** merge (l1 l2:list) := maux l1 l2 (length l1 + length l2).

**Lemma** merge_prop : forall l1 l2 n, length l1 + length l2 <= n \Ra
maux l1 l2 n = merge l1 l2.

The number of reduction steps of \( \text{maux} \ l_1 \ l_2 \ n \) depends only of \( l_1 \) and \( l_2 \).
Minimality

Fixpoint min (p:nat->bool) : nat :=
    if p 0 then 0 else S (min (fun x => p (S x)))

Terminates if there is \( n \) s.t. \( p \ n = \text{true} \).

Fixpoint min2 (p:nat->bool) (n:nat) : nat :=
    if p 0 then 0 else
        match n with
            O => O (* absurd case if \( p \ n = \text{true} \) *)
        | S m => S (min2 (fun x => p (S x)) m)
        end.

Lemma min2min : forall n k, k < min2 p n -> p k = false.
Lemma min2p : forall n, p n = true -> p (min2 p n) = true.

The integer bounds the number of reductions.
General recursion

- Well-founded relation $R$ (no infinite decreasing chain)
- Check that recursive calls in $f\ t$ are on terms $u$ s.t. $Ru\ t$.
- Structural fixpoints correspond to the well-foundation of the subterm relation.
Accessibility

Inductive Acc (x:A) : Prop :=
  Acc_intro : (forall y, R y x -> Acc y) -> Acc x.

Definition well_founded : Prop := forall x, Acc x.

- Acc correspond to a well-founded tree (each branch has a finite length) with arbitrary branching.
- A node of type Acc x has sons for each y s.t. R y x.
- If p : Acc x and q : R y x then
  \text{accR } q := \text{match } p \text{ with Acc_intro } h \Rightarrow h y q \text{ end is of type Acc } y \text{ and structurally smaller than } p.
- \text{f } x := \ldots(f u_1)\ldots(f u_n)\ldots \text{ is coded as}
- \text{f } x (p : \text{Acc } x) := \ldots(f u_1 (\text{accR}_R u_1 x))\ldots(f u_n (\text{accR}_R u_n x))\ldots
Recursive functions

Well-founded recursion

Reductions

Variable B : Type.
Variable F : forall x, (forall y, R y x -> B) -> B.
Fixpoint facc (x:A) (p:Acc x) {struct p} : B :=
  F x (fun y (q:R y x) => facc y (accR q)).
Variable rwf : well_founded.
Definition f (x:A) : B := facc x (rwf x).

▶ Reduction of facc x (Acc_intro h) to
  F x (fun y (q : R y x) => facc y (h y q))

▶ p : Acc x is of type Prop and is thus erased by extraction.

▶ Proofs of Acc x are often opaques and do not reduce . . .

▶ Use \( \forall p : \text{Acc} x, p = \text{Acc_intro} (\text{fun} y (q : R y x) \Rightarrow \text{accR} q) \)
  proved by dependent elimination on p.

▶ Proof of \( f x = F x (\text{fun} y q \Rightarrow f y) \), extentionality hypothesis of F:
  \( \forall x f g, (\forall y (p : R y x), f y p = g y p) \Rightarrow F x f = F x g \)
Minimality

- \( x < y := p y = \text{false} \land x = S y \)
  
  a path from \( x \) is finite if there exists \( y \) s.t. \( x \leq y \land p y = \text{false} \)

- Ad-hoc definition

  \[
  \text{Inductive bound } (p : \text{nat} \rightarrow \text{bool}) : \text{Prop} :=
  \begin{align*}
  & \quad \text{bound0 } : p \; 0 = \text{true} \rightarrow \text{bound } p \\
  & \quad | \text{boundS } : \text{bound } (\text{fun } x \Rightarrow p \; (S \; x)) \rightarrow \text{bound } p.
  \end{align*}
  \]

- Predecessor :

  \[
  \text{Definition boundP } p \; (bp : \text{bound } p) \; (H : p \; 0 = \text{false}) : \text{bound } (\text{fun } x \Rightarrow p \; (S \; x)) :=
  \begin{align*}
  & \quad \text{match } bp \text{ with}
  \begin{align*}
  & \quad \text{bound0 } H1 \Rightarrow \text{match } \text{true_false } (p \; 0) \; H1 \; H \text{ with end}
  \end{align*}
  \]}

- Fixpoint :

  \[
  \text{Fixpoint min } (p : \text{nat} \rightarrow \text{bool}) \; (bp : \text{bound } p) : \text{nat} :=
  \begin{align*}
  & \quad \text{match } \text{bool_dec } (p \; 0) \; \text{false with}
  \begin{align*}
  & \quad \text{right } _ \Rightarrow 0
  \end{align*}
  \]}

B. Barras, C. Paulin (INRIA, UPS)
Plan

Inductive types

Introduction

Recursive functions
  Structural recursion
  Generalised recursion
  Well-founded recursion

Partial functions

Coinductive definitions
  Principles
  Streams
Partial functions

Function $f : A \rightarrow B$ defined on only a subdomain $D$ of $A$.

- Return a default value in $B$ for $x \notin D$
  - Arbitrary if $B$ is a variable : head of list
- Modify the return type: $\text{option } B$.

\[
\text{Inductive option : Type :=}
\begin{align*}
\text{Some : } & B \rightarrow \text{option} \mid \text{None : option.}
\end{align*}
\]

- The program tests whether the input is inside the domain
- Similar to exceptions
- $\forall x, D x \Rightarrow g x = \text{Some}(f x)$.

- Extra argument of domain: $\forall x, x \in D \rightarrow B$
  - Argument erased by extraction: $D : A \rightarrow \text{Prop}$.
  - Proof irrelevance : $f x d_1 = f x d_2$
Plan

Inductive types

Introduction

Recursive functions
  Structural recursion
  Generalised recursion
  Well-founded recursion

Partial functions

Coinductive definitions
  Principles
  Streams
Principles

- Type (or family of types) defined by its constructors
- Values (closed normal term) begins with a constructor
  Construction by pattern-matching \((\text{match} \ldots \text{with} \ldots \text{end})\)
- Biggest fixpoint \(\nu X. F X\) : infinite objects
  - Co-iteration: \(\forall X, (X \subseteq FX) \rightarrow X \subseteq \nu X. F X\)
  - Co-recursion: \(\forall X, (X \subseteq F (X + \nu X.FX)) \rightarrow X \subseteq \nu X.FX\)
  - Co-fixpoint: \(f := H(f) : \nu X.FX\)
    Recursive calls on \(f\) are guarded by the constructors of \(\nu X.FX\).
Example: streams

Variable A : Type.
CoInductive Stream : Type := Cons : A -> Stream -> Stream.
Definition hd (s:Stream) : A := match s with Cons a _ => a end.
Definition tl (s:Stream) : Stream := match s with Cons a t => t end.
CoFixpoint cte (a:A) := Cons a (cte a).
Lemma cte_hd : forall a, hd (cte a) = a. trivial.
Lemma cte_tl : forall a, tl (cte a) = cte a. trivial.
Lemma cte_eq : forall a, cte a = Cons a (cte a).
  intros; transitivity (Cons (hd (cte a)) (tl (cte a)));
  trivial.
  case (cte a); auto.
Function not well guarded

Filter on stream

Variable \( p : A \rightarrow \text{bool} \).

CoFixpoint \( \text{filter} \) \((s : \text{Stream}) : \text{Stream} := \)
  \( \text{if } p \ (\text{hd} \ s) \ \text{then Cons} \ (\text{hd} \ s) \ (\text{filter} \ (\text{tl} \ s)) \)
  \( \text{else filter} \ (\text{tl} \ (p \ s)) \)

Might introduce a closed term of type \( \text{Stream} \) which do not reduce to a constructor.
Coinductive family

Notion of infinite proof:

CoFixpoint cte2 (a:A) := Cons a (Cons a (cte2 a)).

How to prove $\text{cte } a = \text{cte2 } a$?

Extentional equality:

CoInductive eqS (s t:Stream) : Prop :=
    eqS_intros : hd s = hd t -> eqS (tl s) (tl t)
    -> eqS s t.

Proof

CoFixpoint cte_p1 a : eqS (cte a) (cte2 a) :=
    eqS_intro (refl a) (cte_p2 a)
with cte_p2 a : eqS (cte a) (Cons a (cte2 a)) :=
    eqS_intro (refl a) (cte_p1 a).