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 ?

Discriminable constructors

How to prove `true` $\neq$ `false` ?

```
Definition b2p (b:bool) : Type :=
  match b with true => True | false => False end.

Definition true_false (p:true=false) : False :=
  match p in _ = x return b2p x with
  | refl => I end.
```

Discriminable constructors

- `C x1 ... xn <> D y1 ... ym`

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

Plan

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.

$\neq$ HOL or set theory

- Base typed ι 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.

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 \/ b=false
```

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

Programming language vs logical language

- Incomplete pattern-matching

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

`negb 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

Fixpoint

$\texttt{fix}\, f \; : B := t$

- 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 $\texttt{fix}\, f \; : B := t \longrightarrow t[f := \texttt{fix}\, f \; : B := t]$ only in expressions $\texttt{fix}\, f \; : B := t\; y_1 \ldots y_n$ where y_n begins with a constructor.

Enough to encode structural recusion operator.

Fixpoint reduction

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

```
Fixpoint R (n:nat) : C :=
  match n with O => x | S m => f m (R m) end.
```

- We want:
  - $R\,0 \longrightarrow x$
  - $R\,(\mathtt{S}\;m) \longrightarrow f\;m\,(R\;m)$
- Lose normalisation if:
  $R\;n \longrightarrow \mathtt{match}\;n\;\mathtt{with}\;0 \Rightarrow x \mid (\mathtt{S}\;m) \Rightarrow f\;m\,(R\;m)\;\mathtt{end}.$

Defining general recursive functions

`fix` $f\ x : C := t$ with non structural recursive calls.

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

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

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) l2 end.
```

Reductions:

- $\texttt{merge}\ []\ l_2 \Longrightarrow l_2$
- $\texttt{merge}\ (a :: m_1) \Longrightarrow \texttt{fix maux}\ m := \texttt{match}\ m\ \texttt{with}\ [] \Rightarrow a :: m_1$
  $\mid b :: m_2 \Rightarrow \texttt{if}\ a < b\ \texttt{then}\ a :: \texttt{merge}\ m_1\ m$
  $\texttt{else}\ b :: \texttt{maux}\ m_2\ \texttt{end})$
- $\texttt{merge}\ (a :: m_1)\ [] \Longrightarrow a :: m_1$
- $\texttt{merge}\ (a :: m_1)\ (b :: m_2) \equiv$
  $\texttt{if}\ a < b\ \texttt{then}\ a :: \texttt{merge}\ m_1\ (b :: m_2)$
  $\texttt{else}\ b :: \texttt{merge}\ (a :: m_1)\ m_2$

Encoding recursion

Extra argument: measure (length of $l_1@l_2$).

```
Fixpoint maux (l1 l2:list) (n:nat) : list :=
  match (l1,l2) with
     []  , _, n => l2 | _  , [], n => l1
   | _   , _, 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 `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. $p\,n = \texttt{true}$.

```
Fixpoint min2 (p:nat->bool) (n:nat) : nat :=
  if p 0 then 0 else
  match n with O => O (* absurd case if p n = 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. $R\,u\,t$.
- 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 `accR` $q :=$ `match` p `with Acc_intro` $h \Rightarrow h\ y\ q$ `end` is of type `Acc` y and structurally smaller than p.
- $f\ x := \ldots (f\ u_1) \ldots (f\ u_n) \ldots$ is coded as
- $f\ x\ (p : \texttt{Acc}\ x) := \ldots (f\ u_1\ (\texttt{accR}\ ?_{R\,u_1\,x})) \ldots (f\ u_n\ (\texttt{accR}\ ?_{R\,u_n\,x})) \ldots$

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 $\texttt{facc}\, x\, (\texttt{Acc\_intro}\, h)$ to $F\ x\, (\texttt{fun}\ y\, (q : R\, y\, x) \Rightarrow \texttt{facc}\, y\, (h\, y\, q))$
- $p : \texttt{Acc}\, x$ is of type $\texttt{Prop}$ and is thus erased by extraction.
- Proofs of $\texttt{Acc}\, x$ are often opaques and do not reduce ...
- Use $\forall p : \texttt{Acc}\, x, p = \texttt{Acc\_intro}\, (\texttt{fun} y\ (q : R\, y\, x) \Rightarrow \texttt{accR}\ q)$ proved by dependent elimination on p.
- Proof of $f\, x = F\, x\, (\texttt{fun}\ y\ q \Rightarrow f\ y)$, 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

- $x \prec y := p\,y = \texttt{false} \wedge x = \texttt{S}\,y$
  a path from x is finite if there exists y s.t. $x \leq y \wedge p\,y = \texttt{false}$
- Ad-hoc definition

```
Inductive bound (p:nat->bool) : Prop :=
   bound0 : p 0 = true -> bound p
 | boundS : bound (fun x => p (S x)) -> bound p.
```

- Predecessor :

```
Definition boundP p (bp: bound p) (H:p 0 = false)
  : bound (fun x => p (S x)) :=
 match bp with
   bound0 H1 => match true_false (p 0) H1 H with end
 | boundS bp' => bp' end.
```

- Fixpoint :

```
Fixpoint min (p:nat->bool) (bp:bound p) : nat :=
 match bool_dec (p 0) false with  right _ => 0
 | left H => S(min(fun x=>p(S x)) (boundP p bp H))end.
```

Plan

Partial functions

Function $f : A \to B$ 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: `option` B.

```
Inductive option:Type :=
  Some : B -> option | None : option.
```

  - The program tests whether the input is inside the domain
  - Similar to exceptions
  - $\forall x, D\,x \Rightarrow g\,x = \texttt{Some}\,(f\,x)$.
- Extra argument of domain: $\forall x, x \in D \to B$
  - Argument erased by extraction: $D : A \to Prop$.
  - Proof irrelevance : $f\,x\,d_1 = f\,x\,d_2$

Plan

Principles

- Type (or family of types) defined by its constructors
- Values (closed normal term) begins with a constructor
  Construction by pattern-matching (`match...with...end`)
- Biggest fixpoint $\nu X.F\,X$: infinite objects
  - Co-iteration: $\forall X, (X \subseteq FX) \to X \subseteq \nu X.F\,X$
  - Co-recursion: $\forall X, (X \subseteq F\,(X + \nu X.FX)) \to X \subseteq \nu X.FX$
  - 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->bool.
CoFixpoint filter (s:Stream) : Stream :=
  if p (hd s) then Cons (hd s) (filter (tl s))
  else filter (tl (p s))
```

Might introduce a closed term of type `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 `cte` a = `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).
```