Inductive Constructions

TYPES Summer School, Bertinoro, Italy

Christine Paulin-Mohring

INRIA Futurs & Université Paris Sud

August 2007

Inductive Constructions: outline

Material for the course

http://www.lri.fr/~paulin/TypesSummerSchool

Course 1 : Basic notions

- Introduction
- Inductive constructions in practice
- Encoding in HOL
- Rules for COQ inductive constructions

Inductive Constructions: outline

Course 2 : Advanced notions

- Equality
- Paradoxes
- Coinductive definitions
- Extensions

Part I

Inductive Constructions : basic notions

Plan

Informal definition

An inductive definition introduces a new set of objects (predicate) I by :

- a set of rules of constructions for object in I (proofs of I).
- initiality : If T admits the same rules of constructions then $I \subseteq T$
  - smallest notion closed under the rules of constructions
  - distinct rules of constructions give distinct objects

Informal definition

An inductive definition introduces a new set of objects (predicate) I by :

- a set of rules of constructions for object in I (proofs of I).
- initiality : If T admits the same rules of constructions then $I \subseteq T$
  - smallest notion closed under the rules of constructions
  - distinct rules of constructions give distinct objects

Informal definition

An inductive definition introduces a new set of objects (predicate) I by :

- a set of rules of constructions for object in I (proofs of I).
- initiality : If T admits the same rules of constructions then $I \subseteq T$
  - smallest notion closed under the rules of constructions
  - distinct rules of constructions give distinct objects

Inductive definitions are everywhere !

- programming
  - data structures : enumerated types, records, sum natural numbers, lists, trees . . .
  - clauses in logic programming : predicate definition
- semantics of programming languages
  - abstract syntax trees
  - inference rules for static or operational semantics
- logic
  - representation of terms, formulas (grammars)
  - semantics, deduction relation
  - constructive interpretation of connectors Curry-Howard isomorphism
- proof assistant
  - basic notion in Martin-Löf's Type Theory (Agda)
  - encoded in HOL
  - primitive in COQ but some can also be encoded

Inductive definitions are everywhere !

- programming
  - data structures : enumerated types, records, sum
    natural numbers, lists, trees . . .
  - clauses in logic programming : predicate definition
- semantics of programming languages
  - abstract syntax trees
  - inference rules for static or operational semantics
- logic
  - representation of terms, formulas (grammars)
  - semantics, deduction relation
  - constructive interpretation of connectors
    Curry-Howard isomorphism
- proof assistant
  - basic notion in Martin-Löf's Type Theory (Agda)
  - encoded in HOL
  - primitive in COQ but some can also be encoded

Inductive definitions are everywhere !

- programming
  - data structures : enumerated types, records, sum natural numbers, lists, trees . . .
  - clauses in logic programming : predicate definition
- semantics of programming languages
  - abstract syntax trees
  - inference rules for static or operational semantics
- logic
  - representation of terms, formulas (grammars)
  - semantics, deduction relation
  - constructive interpretation of connectors Curry-Howard isomorphism
- proof assistant
  - basic notion in Martin-Löf's Type Theory (Agda)
  - encoded in HOL
  - primitive in COQ but some can also be encoded

Inductive definitions are everywhere !

- programming
  - data structures : enumerated types, records, sum
    natural numbers, lists, trees . . .
  - clauses in logic programming : predicate definition
- semantics of programming languages
  - abstract syntax trees
  - inference rules for static or operational semantics
- logic
  - representation of terms, formulas (grammars)
  - semantics, deduction relation
  - constructive interpretation of connectors
    Curry-Howard isomorphism
- proof assistant
  - basic notion in Martin-Löf's Type Theory (Agda)
  - encoded in HOL
  - primitive in COQ but some can also be encoded

Two different views

Mathematics

- Sets as primitive objects
- Natural numbers, relations, functions as derived notions
- Extensional equality

Programming language or proof assistant

- Every constructions should be justified, implemented
- Intensional view of objects
- Functions as algorithms
- Computation

Two different views

Mathematics

- Sets as primitive objects
- Natural numbers, relations, functions as derived notions
- Extensional equality

Programming language or proof assistant

- Every constructions should be justified, implemented
- Intensional view of objects
- Functions as algorithms
- Computation

Two different views

Mathematics

- Sets as primitive objects
- Natural numbers, relations, functions as derived notions
- Extensional equality

Programming language or proof assistant

- Every constructions should be justified, implemented
- Intensional view of objects
- Functions as algorithms
- Computation

Inductive definitions in proof-assistants

Representation

- encoded
- primitive notion in the theory

Which class of inductive definitions ?

- (strictly) positive, monotonic, no restriction
- polymorphic, impredicative . . .
- mutually inductive definitions, inductive families . . .

Which rules ?

- primitive rules / derived rules
  - pattern-matching
  - primitive recursion
  - course of value recursion
  - . . .

Inductive definitions in proof-assistants

Representation

- encoded
- primitive notion in the theory

Which class of inductive definitions ?

- (strictly) positive, monotonic, no restriction
- polymorphic, impredicative ...
- mutually inductive definitions, inductive families ...

Which rules ?

- primitive rules / derived rules
  - pattern-matching
  - primitive recursion
  - course of value recursion
  - ...

Inductive definitions in proof-assistants

Representation

- encoded
- primitive notion in the theory

Which class of inductive definitions ?

- (strictly) positive, monotonic, no restriction
- polymorphic, impredicative ...
- mutually inductive definitions, inductive families ...

Which rules ?

- primitive rules / derived rules
  - pattern-matching
  - primitive recursion
  - course of value recursion
  - ...

Encoded inductive definitions

- To a set I corresponds a type $\widehat{I}$
- To $t \in I$ corresponds a term $\hat{t} : \widehat{I}$
- To a property $P\,t$ corresponds a proof $\vdash \widehat{P\,t}$

Question Adequation of the representation ?

Example

- $\mathbb{N}$ encoded as $\forall\alpha, (\alpha \to \alpha) \to \alpha \to \alpha$
- $n \in \mathbb{N}$ encoded as $\widehat{n} \equiv \lambda\alpha\, f\, x, f^n\, x$
  $n + m \equiv \lambda\alpha\, f\, x, n\, \alpha\, f\, (m\, \alpha\, f\, x)$
- $\forall x, 0 + x = x$, $\widehat{n} + 0 = \widehat{n}$
- $0 \neq 1$ not provable in pure Calculus of Constructions

Primitive inductive definitions

Basic principles

- Which class of inductive definitions ?
- Primitive eliminations ?
  recursive combinators, pattern-matching, fixpoints, rewriting ...

Good computational behavior

- Termination of computations
- Extraction, proof irrelevance

Consistency of the underlying logic

Goal of the course

Better understanding of Inductive Definitions

- How to use them ?
- Intuition on their power.
- Explanation of some design choices.
- What can/cannot be done with inductive definitions.

Notations

inspired by Coq notations, informal in the first part

- sorts: Prop, Type, $*$
- function space:
  - $\forall (x : A), B$
  - $A \rightarrow B$
  - $\forall x, B \;\; \forall (x_1 : A_1)..(x_n : A_n), B.$
- abstraction: **fun** $x \Rightarrow t$
- application: $t\, x_1 \ldots x_n$

Outline

Outline

Basic (non-recursive) data types

Concrete data-types/propositional logic

- enumerated sets (absurdity, truth)

```
Inductive empty := . (* False *)
Inductive unit := tt. (* True *)
Inductive bool := true | false
```

- disjoint sum, constructive disjunction

```
Inductive sum A B    (* or A B *)
    := inl : A → sum A B
     | inr : B → sum A B.
```

Basic (non-recursive) data types

Concrete data-types/propositional logic

- enumerated sets (absurdity, truth)

```
Inductive empty := . (* False *)
Inductive unit := tt. (* True *)
Inductive bool := true | false
```

- disjoint sum, constructive disjunction

```
Inductive sum A B    (* or A B *)
    := inl : A → sum A B
     | inr : B → sum A B.
```

Basic (non-recursive) data types

Concrete data-types/propositional logic

- enumerated sets (absurdity, truth)

```
Inductive empty := . (* False *)
Inductive unit := tt. (* True *)
Inductive bool := true | false
```

- disjoint sum, constructive disjunction

```
Inductive sum A B    (* or A B *)
    := inl : A → sum A B
     | inr : B → sum A B.
```

Product

- product, conjunction

```
Inductive prod A B (* and A B *)
    := pair : A → B → prod A B.
```

- dependent product, existential quantifier

```
Inductive sig (P : A → *) (* ex x:A, P *)
    := sigi : ∀ x:A, P x → sig P.
```

a ∀ in the type of the constructor corresponds to a constructive existential quantification.

General pattern

```
Inductive I (params)
   := ...
   | c_i : ∀ x_1:A_1 ... x_n:A_n, I params
   | ...
```

- Non-recursive case: I does not occur in A_i
- Property: any $x : I$ is of the form $c_i\, a_1 \ldots a_n$
- Elimination: complete pattern-matching

```
f : empty → C := .
f : unit → C := f tt ⇒ x.
f : bool → C := f true ⇒ x | f false ⇒ y.
f : sum A B → C :=
    f (inl a) ⇒ x | f (inr b) ⇒ y.
```

Recursive data types

Algebraic datatypes: natural numbers, lists, trees ...
Pattern-matching + structural recursion

```
Inductive nat := O | S : nat → nat.
Def f : nat → C :=
    f O ⇒ ...
    f (S n) ⇒ ... (f n) ...
```

Induction principle

$$\forall P, P\, O \rightarrow (\forall n, P\, n \rightarrow P\,(S\, n)) \rightarrow \forall n, P\, n$$

Exercice

```
Inductive I := c: I → I.
```

- Write the induction principle associated to I
- Give the general form of primitive recursive functions of type $I \to C$
- I is empty: build a function of type $I \to empty$

More recursive data types

Trees with denumerable branching.

```
Inductive ord : Type :=
  zero | succ : ord → ord
| lim : (nat → ord) → ord.

Def nat2ord : nat → ord :=
    O ⇒ zero
| S n ⇒ succ (nat2ord n).
```

(`lim nat2ord`) is infinite but each branch is finite.

Induction principle:

```
∀ P, P zero → (∀ x, P x → P (succ x))
   → (∀ f, (∀ n,P (f n)) → P (lim f))
   → ∀ x:ord, P x
```

Well-founded type

$W x : A . B\, x$

Each node is parametrized by $a : A$
with subtrees indexed by $b : B\, a$.

```
Inductive W (B:A→ ⋆)
   := Node : ∀ a:A, (B a → W B) → W B.
```

Recursive calls on subtrees:

```
Def f : W B → C :=
  (Node a t) ⇒...(f (t b1))...(f (t bp))...
```

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- Encoding in HOL
- Rules for COQ inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Predicate definition

Logic

- $x = y$ with introduction rule: $x = x$

Logic programming

```
le O n :-
le (S n) (S m) :- le n m
```

Semantics

$$\frac{s \vdash b \leadsto \texttt{true} \quad s \vdash p \leadsto s_1 \quad s_1 \vdash (\texttt{while}\, b\, p) \leadsto s_2}{s \vdash (\texttt{while}\, b\, p) \leadsto s_2}$$

$$\frac{s \vdash b \leadsto \texttt{false}}{s \vdash (\texttt{while}\, b\, p) \leadsto s}$$

Example

Reflexive-transitive closure of R

```
Inductive RT :=
  RTrefl:∀ x, RT x x
| RTR:∀ x y, R x y → RT x y
| RTtran:∀ x y z, RT x z → RT z y → RT x y.
```

Fixpoint:

$$RT\,x\,y \leftrightarrow x = y \lor R\,x\,y \lor (\exists z, RT\,x\,z \land RT\,z\,y)$$

Minimality:

$$\begin{array}{ll}\forall P, & (\forall x, P\,x\,x) \\ & \rightarrow (\forall xy, R\,x\,y \rightarrow P\,x\,y) \\ & \rightarrow (\forall xyz, RT\,x\,y \rightarrow P\,x\,y \rightarrow RT\,y\,z \rightarrow P\,y\,z \rightarrow P\,x\,z) \\ & \rightarrow \forall xy, RT\,x\,y \rightarrow P\,x\,y\end{array}$$

Exercise

- Prove $RT\,x\,y \leftrightarrow x = y \vee \exists z, RT\,x\,z \wedge R\,z\,y$
- Write a new inductive definition of the transitive-closure corresponding to the previous equivalence
- Write the corresponding minimality principle

General pattern

```
Inductive I := ...
  | c_i : ∀ x_1:A_1...x_n:A_n, I t_1..t_p
  | ... .
```

$t_1 \ldots t_p$ are arbitrary terms,
not only parameters (distinct quantified variables).
Recursive arguments on other instances $I\ u_1..u_p$

Smallest relation preserving the rules of construction.

Equality is one of the central notions:

```
Inductive I y_1..y_p := ...
  | c_i : ∀x_1:A_1...x_n:A_n,
               t_1..t_p=y_1..y_p → I y_1..y_p
  | ...
```

Equality

Smallest reflexive relation:

```
Inductive eq := eqrefl : ∀ x, eq x x.
```

Minimality:

$$\forall P, (\forall x, P\,x\,x) \rightarrow \forall xy, eq\,x\,y \rightarrow P\,x\,y$$

Different elimination (Leibniz equality):

$$\forall Q\,x\,y, Q\,x \rightarrow eq\,x\,y \rightarrow Q\,y$$

Derivable equivalence

- 2 $\Rightarrow$ 1 Given $P\ x$, takes $Q \equiv P\,x$
- 1 $\Rightarrow$ 2 Given Q takes $P\,x\,y \equiv Q\,x \rightarrow Q\,y$

2 is always easier to use.

Parameters

corresponds to an outside quantification:

```
Variable x.
Inductive eq_x := refleq_x : eq_x x.
```

usually written:

```
Inductive eq x := eqrefl : eq x x.
```

Generated minimality principle:

$$\forall x, \forall Q, Q\, x \rightarrow \forall y, eq\, x\, y \rightarrow Q\, y$$

General pattern:

```
Inductive I (params) := ...
  | c_i : ∀ x_1:A_1...x_n:A_n, I params t_1..t_p
  | ... .
```

Inversion

The conclusion of the minimality principle is
$\forall x_1 \ldots x_p, I\, x_1 \ldots x_p \rightarrow P\, x_1 \ldots x_p$
P should be true for all instances of I.
Sometimes, we want to prove special instances:

- $le\,(S\,n)\,0 \rightarrow False$
- $le\,(S\,n)\,(S\,m) \rightarrow le\,n\,m$

Advanced pattern-matching rules, or find a generalization:

- $le\,x\,y \rightarrow x = S\,n \rightarrow y = 0 \rightarrow False$
  inversion
- $le\,x\,y \rightarrow le\,(P\,x)\,(P\,y)$

Inversion

The conclusion of the minimality principle is
$\forall x_1 \ldots x_p, I\, x_1 \ldots x_p \to P\, x_1 \ldots x_p$
P should be true for all instances of I.
Sometimes, we want to prove special instances:

- $le\,(S\,n)\,0 \to False$
- $le\,(S\,n)\,(S\,m) \to le\,n\,m$

Advanced pattern-matching rules, or find a generalization:

- $le\,x\,y \to x = S\,n \to y = 0 \to False$
  inversion
- $le\,x\,y \to le\,(P\,x)\,(P\,y)$

Inversion

The conclusion of the minimality principle is
$\forall x_1 \ldots x_p, I\, x_1 \ldots x_p \rightarrow P\, x_1 \ldots x_p$
P should be true for all instances of I.
Sometimes, we want to prove special instances:

- $le\,(S\,n)\,0 \rightarrow False$
- $le\,(S\,n)\,(S\,m) \rightarrow le\,n\,m$

Advanced pattern-matching rules, or find a generalization:

- $le\,x\,y \rightarrow x = S\,n \rightarrow y = 0 \rightarrow False$
  inversion
- $le\,x\,y \rightarrow le\,(P\,x)\,(P\,y)$

Different definitions of the order on natural numbers

Recursive definition

Def le0 : nat → nat → prop
  le_0 O n ⇒ True | le_0 (S n) O ⇒ False
 | le_0 (S n) (S m) ⇒ le_0 n m.

Inductive definitions:

Inductive le_1 : nat → nat → prop
  le_1b : ∀ n, le_1 O n
 | le_1S : ∀ n m, le_1 n m → le_1 (S n) (S m).

Inductive le_2 (n : nat): nat → prop
  le_2b : le_2 n n
 | le_2S : ∀ m, le_2 n m → le_2 n (S m).

Different definitions of the order on natural numbers

Recursive definition

```
Def le0 : nat → nat → prop
   le0 O n ⇒ True | le0 (S n) O ⇒ False
 | le0 (S n) (S m) ⇒ le0 n m.
```

Inductive definitions:

```
Inductive le1 : nat → nat → prop
   le1b : ∀ n, le1 O n
 | le1S : ∀ n m, le1 n m → le1 (S n) (S m).
```

```
Inductive le2 (n : nat): nat → prop
   le2b : le2 n n
 | le2S : ∀ m, le2 n m → le2 n (S m).
```

Different definitions of the order on natural numbers

Recursive definition

Def le0 : nat → nat → prop
le_0 O n ⇒ True | le_0 (S n) O ⇒ False
| le_0 (S n) (S m) ⇒ le_0 n m.

Inductive definitions:

Inductive le_1 : nat → nat → prop
le_1b : ∀ n, le_1 O n
| le_1S : ∀ n m, le_1 n m → le_1 (S n) (S m).

Inductive le_2 (n : nat): nat → prop
le_2b : le_2 n n
| le_2S : ∀ m, le_2 n m → le_2 n (S m).

Equivalence of definitions

	le_0	le_1	le_2
$2 \leq 3$	*true*	$le_1\, S\,(le_1\, S\,(le_1\, b\, 1))$	$le_2\, S\,(le_2\, b\, 2)$
$0 \leq n$	*true*	$le_1\, b$	*ind n*
$n \leq m \Rightarrow S\,n \leq S\,m$			
	$A \Rightarrow A$	$le_1\, S$	*ind* $(n \leq m)$
$n \leq n$	*ind n*	*ind n*	$le_2\, b$
$n \leq m \Rightarrow n \leq S\,m$			
	double ind	*ind* $(n \leq m)$	$le_2\, S$
$S\,n \leq S\,m \Rightarrow n \leq m$			
	$A \Rightarrow A$	*inversion*	*hard*
$S\,n \not\leq O$	$\neg$*false*	*inversion*	*inversion*
transitive	*double ind*	*double ind*	*ind*$(m \leq p)$

$le_1 \Leftrightarrow le_2$ $\qquad$ $le_0 \Rightarrow le_1$ (*double ind*)

Remarks

- Recursive definitions are not always possible (see *RT*)
- Inductive definitions are not always possible (positivity)
  - $SN\,x \rightarrow x \in [base]$
  - $(\forall x, x \in [\sigma] \rightarrow (app\;t\;x) \in [\tau]) \rightarrow t \in [arr\;\sigma\;\tau]$
- Possible choice between different inductive/recursive specifications
  (some theorems for free, other more complicated to get)

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- Encoding in HOL
- Rules for Coq inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Inductive families

Proof of inductively defined relation seen as concrete objects

```
Inductive list :=
   nil : list O
| cons : ∀ n, A → list n → list (S n)
```

Pattern-matching and recursive definitions as for lists.
Induction principle:

```
∀ P, (P O nil)
   → (∀ n a l, P n l → P (S n) (cons n a l))
   → ∀ n l, P n l
```

Concrete view of relations

Finite sets

```
inductive finite : set → ⋆ :=
   fin0 : finite ∅
 | finadd : ∀ a s, finite s → a∉ s → finite ({a}∪ s)
```

The proof is a list enumerating the elements without duplication.

Derivations in minimal logic

```
inductive pr : list form → form → ⋆ :=
  elim: ∀ E A B, pr E (A⇒B) → pr E A → pr E B
| intro: ∀ E A B, pr (A::E) B → pr E (A⇒B)
| var: ∀ E A, pr (A::E) A
| weak: ∀ E A B, pr E B → pr (A::E) B.
```

A proof of *pr E A* encodes a lambda-term.

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- Encoding in HOL
- Rules for COQ inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Structural recursion

```
Def f : nat → C :=
    f O ⇒ ...
    f (S n) ⇒ ... (f n) ...
```

like primitive recursion but *C* can be functional

```
Def ackf (f:nat → nat) : nat →  nat :=
     ackf f O ⇒ f (S O)
     ackf f (S m) ⇒ f (ackf f m).
Def ack : nat → nat → nat :=
     ack O ⇒ (fun m ⇒ m)
     ack (S n) ⇒ ackf (ack n)
```

A theoretical result

Any recursive function, provably total in (higher-order) arithmetic can be represented using (higher-order) primitive recursive scheme.

- Take a recursive function f
- Kleene T,U (primitive recursive) : natural number n which represents f : $f(x) = y \Leftrightarrow \exists k.T\,n\,x\,k \wedge U\,k = y$
- f is provably total if there is a proof of : $\forall x.\exists k.T\,n\,x\,k$
- From this proof, one can extract a (higher-order) primitive recursive representation.

Maybe not the appropriate algorithm !

A theoretical result

Any recursive function, provably total in (higher-order) arithmetic can be represented using (higher-order) primitive recursive scheme.

- Take a recursive function f
- Kleene T,U (primitive recursive) : natural number n which represents f : $f(x) = y \Leftrightarrow \exists k.T\,n\,x\,k \wedge U\,k = y$
- f is provably total if there is a proof of : $\forall x.\exists k.T\,n\,x\,k$
- From this proof, one can extract a (higher-order) primitive recursive representation.

Maybe not the appropriate algorithm !

Schemes

```
Def f (n:nat) : C := h (f n1)...(f np).
```

with $n_i < n$.

```
Def g : nat → nat → C :=
  g n O ⇒ ? | g n (S m) ⇒ h (g n1 m)...(g np m).
Def f (n:nat) : C := g n (S n).
```

Properties:

```
Lemma ∀ m n, n<m → f n = g n m.
Lemma ∀ n, f n = h (f n1)...(f np).
```

Associated induction principle:

$$\forall P, (\forall n, (\forall m, m < n \rightarrow P\, m) \rightarrow P\, n) \rightarrow \forall n, P\, n$$

Well-founded ordering

Relation $<$ with no infinite chain

```
Inductive acc x
   := acci : (∀ y, y<x → acc y) → acc x.
```

$<$ is well-founded if $wf : \forall x, acc\,x$

Structural recursion:

```
F : ∀ x, acc x → C :=
  F x (acci t) ⇒ ... (F y (t y ?y<x)) ...
```

Induction principle:

$$\forall P, (\forall y, y < x \rightarrow P\,y) \rightarrow P\,x) \rightarrow \forall x, acc\,x \rightarrow P\,x$$

Well-founded ordering

Relation $<$ with no infinite chain

```
Inductive acc x
   := acci : (∀ y, y<x → acc y) → acc x.
```

$<$ is well-founded if *wf* : $\forall x, acc\,x$

Structural recursion:

```
F : ∀ x, acc x → C :=
  F x (acci t) ⇒ ... (F y (t y ?y<x)) ...
```

Induction principle:

$$\forall P, (\forall y, y < x \rightarrow P\,y) \rightarrow P\,x) \rightarrow \forall x, acc\,x \rightarrow P\,x$$

Well-founded recursion

Def `f (x : A) : h (f` x_1`)...(f` x_p`).`

with $x_i < x$ for a well-founded order $<$.

Def `g (x : A) (a : acc x) : C :=`
`g x (acci t)` $\Rightarrow$ `h...(g` x_i `(t` x_i $?_{x_i<x}$`))...`

Def `f (x : A) := g x (wf x)`

Property :

Lemma $\forall$ `(x : A) (a : acc x), f x = g x a`
Lemma $\forall$ `(x : A) f x = h (f` x_1`)...(f` x_p`).`

Minimisation

```
min P n = if P n then n else min P (S n)
```

introduce $x \prec y := \neg P\, y \wedge x = S\, y$
show $\forall m, P\, m \to acc_{\prec}\, 0$

Remark: we cannot expect a (strongly normalizing) reduction

$$min\, P\, n \longrightarrow \textbf{if}\ P\, n\ \textbf{then}\ n\ \textbf{else}\ min\, P\, (S\, n)$$

Termination arguments

```
Inductive prog :=
    Base : (state → state) → prog
  | Seq : prog → prog → prog (* p1;p2 *)
  | While : (state → bool) → prog → prog.
```

How to write an evaluation function ?

```
Inductive E : state→prog→state→Prop :=
   EBase : ∀ s p, E s (Base p) (p s)
 | ESeq : ∀ s1 s2 s3 p q,
    E s1 p s2 → E s2 q s3 → E s1 (p;q) s3
 | EWhiletrue : ∀ s1 s2 b p,
    b s1 = true → E s1 (p;While b p) s2
    → E s1 (While b p) s2
 | EWhilefalse : ∀ s b p,
    b s = false → E s (While b p) s.
```

Termination arguments

```
Inductive prog :=
    Base : (state → state) → prog
  | Seq : prog → prog → prog (* p1;p2 *)
  | While : (state → bool) → prog → prog.
```

How to write an evaluation function ?

```
Inductive E : state→prog→state→Prop :=
   EBase : ∀ s p, E s (Base p) (p s)
 | ESeq : ∀ s1 s2 s3 p q,
    E s1 p s2 → E s2 q s3 → E s1 (p;q) s3
 | EWhiletrue : ∀ s1 s2 b p,
    b s1 = true → E s1 (p;While b p) s2
    → E s1 (While b p) s2
 | EWhilefalse : ∀ s b p,
    b s = false → E s (While b p) s.
```

Terminating programs

```
Inductive T : state → prog → Prop :=
   TBase : ∀ s p, T s (Base p)
 | TSeq : ∀ s p q,
     T s p → (∀ s′, E s p s′ → T s′ q)
     → T s (p;q)
 | TWhiletrue : ∀ s b p,
     b s = true → T s (p;While b p)
     → T s (While b p)
 | TWhilefalse : ∀ s b p,
     b s = false → T s (While b p).
```

Evaluation function

Recursively defined on terminating programs:

```
Def eval : ∀ s p, T s p → ex s', E s p s'
eval s (Base p) (TBase s p) ⇒ (p s,?)
eval s (p1;p2) (TSeq s p1 p2 tp1 tp2) ⇒
      let (s1,e1) := eval s p1 tp1 in
      let (s2,_) := eval s1 p2 (tp2 s1 e1)
      in (s2,?)
eval s (While b p) (TWhiletrue s b p H ts) ⇒
      let (s',_) := eval s (p;While b p) ts
      in (s',?)
eval s (While b p) (TWhilefalse s b p H) ⇒ (s,?)
```

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- **Encoding in HOL**
- Rules for COQ inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Encoding of non-recursive predicates

Inductive I *pars* :=
  ... c_i:∀ $(x_1 : A_1)..(x_n : A_n)$ I *pars* $u_1..u_p$...

is translated into

Def I *pars* $y_1..y_p$:=
  ∀ P, ... (∀ $(x_1 : A_1)..(x_n : A_n)$ P $u_1..u_p$) ...
→ P $y_1..y_p$

- No dependent types, no pattern-matching.
- Works for equality, inference rules ...

Encoding of inductive predicates

Unary predicate with one constructor

```
Inductive I x := c : F I x → I x.
```

F has type $(A \to prop) \to (A \to prop)$
F should be monotonic : $\mathtt{mon} : X \subseteq Y \to F\,X \subseteq F\,Y$
with $X \subseteq Y \equiv \forall x, X\,x \to Y\,x$

Introduce

```
Def I x := ∀ P, (F P ⊆ P) → P x.
```

Introduction/Elimination schemes

```
Def I x := ∀ P, (F P ⊆ P) → P x.
```

Iteration scheme is trivial.

```
Def it := ∀ P, (F P ⊆ P) → I ⊆ P.
```

Constructor:

```
Def c : F I ⊆ I :=
    fun x (t:F I x) P (f : F P ⊆ P) ⇒
    f (mon (it P f) x t).
```

Recursors:

```
Def rec1 := ∀ P, (F (I ∩ P) ⊆ P) → I ⊆ P.
Def rec2 := ∀ P,
   (∀ Q, (Q ⊆ I) → (Q ⊆ P) → F Q ⊆ P)
   → I ⊆ P.
```

Exercise: show that these schemes are equivalent

Inductive types

T. Melham, E. Gunter, L. Paulson, J. Harrison…

The key steps :

- Define a type X, such that one can build injective functions for the constructors. $z : X \quad s : X \to X \quad s\,x = s\,y \to x = y \quad s\,x \neq z$
- Define by induction the smallest subset $I\,x$ of X closed by the rules of construction. $N\,x = \forall P, P\,z \to (\forall y. P\,y \to P\,(s\,y)) \to P\,x$
- Define $\mathtt{I}$ as the restriction of X to objects x which satisfy $I\,x$.
  $\mathbf{abs} : X \to \mathtt{I} \quad \mathbf{rep} : \mathtt{I} \to X$
  $\mathbf{abs}\,(\mathbf{rep}\,n) = n \quad I\,x \to \mathbf{rep}(\mathbf{abs}\,x) = x \quad I\,(\mathbf{rep}\,n)$
  $I\,x \to I\,y \to \mathbf{abs}\,x = \mathbf{abs}\,y \to x = y$

Properties of the inductive type

- Define constructors of $\mathtt{I}$ with the appropriate type using **abs** and **rep**.
  $O = \textbf{abs}\, z \quad S\, n = \textbf{abs}\,(s\,(\textbf{rep}\, n))$
- $S\, n = S\, m \rightarrow n = m$
  because $s\,(\textbf{rep}\, n) = s\,(\textbf{rep}\, m) \rightarrow \textbf{rep}\, n = \textbf{rep}\, m$
- $S\, n \neq O$ because $s\,(\textbf{rep}\, n) \neq z$
- Derive induction principle for $\mathtt{I}$ using property I.

$$\frac{P\, 0 \qquad \forall n : N, P\, n \rightarrow P\,(S\, n)}{\forall n : N, P\, n}$$

  Show $\forall x, N\, x \rightarrow P\,(\textbf{abs}\, x)$

Properties of the inductive type

- Define constructors of $\mathtt{I}$ with the appropriate type using **abs** and **rep**.
  $O = \textbf{abs}\, z \quad S\, n = \textbf{abs}\,(s\,(\textbf{rep}\, n))$
- $S\, n = S\, m \rightarrow n = m$
  because $s\,(\textbf{rep}\, n) = s\,(\textbf{rep}\, m) \rightarrow \textbf{rep}\, n = \textbf{rep}\, m$
- $S\, n \neq O$ because $s\,(\textbf{rep}\, n) \neq z$
- Derive induction principle for $\mathtt{I}$ using property I.

$$\frac{P\, 0 \qquad \forall n : N, P\, n \rightarrow P\,(S\, n)}{\forall n : N, P\, n}$$

  Show $\forall x, N\, x \rightarrow P\,(\textbf{abs}\, x)$

Recursion scheme

Prove the existence of a general recursor

$\forall g : \alpha, \forall h : \mathrm{N} \to \alpha \to \alpha.$
$\exists! f : \mathrm{N} \to \alpha, f\,0 = g \wedge \forall n. f\,(S\,n) = h\,n\,(f\,n)$

Define $F : \mathrm{N} \to \alpha \to o$ inductively :

$$\frac{}{F\,0\,g} \qquad \frac{F\,n\,a}{F\,(S\,n)\,(h\,n\,a)}$$

Prove by induction on n

- $\forall n : \mathrm{N}, \exists a : \alpha, F\,n\,a$
- $\forall (n : \mathrm{N})(a\,b : \alpha), F\,n\,a \to F\,n\,b \to a = b$

Take $f\,n$ be $\epsilon a, F\,n\,a$.
Computation done by equational reasoning.

Recursion scheme

Prove the existence of a general recursor
$\forall g : \alpha, \forall h : \mathbb{N} \rightarrow \alpha \rightarrow \alpha.$
$\exists! f : \mathbb{N} \rightarrow \alpha, f\,0 = g \wedge \forall n. f\,(S\,n) = h\,n\,(f\,n)$
Define $F : \mathbb{N} \rightarrow \alpha \rightarrow o$ inductively :

$$\frac{}{F\,0\,g} \qquad \frac{F\,n\,a}{F\,(S\,n)\,(h\,n\,a)}$$

Prove by induction on n

- $\forall n : \mathbb{N}, \exists a : \alpha, F\,n\,a$
- $\forall (n : \mathbb{N})(a\,b : \alpha), F\,n\,a \rightarrow F\,n\,b \rightarrow a = b$

Take $f\,n$ be $\epsilon a, F\,n\,a$.
Computation done by equational reasoning.

Recursion scheme

Prove the existence of a general recursor
$\forall g : \alpha, \forall h : \mathrm{N} \to \alpha \to \alpha.$
$\exists! f : \mathrm{N} \to \alpha, f\,0 = g \wedge \forall n. f\,(S\,n) = h\,n\,(f\,n)$
Define $F : \mathrm{N} \to \alpha \to o$ inductively :

$$\frac{}{F\,0\,g} \qquad \frac{F\,n\,a}{F\,(S\,n)\,(h\,n\,a)}$$

Prove by induction on n

- $\forall n : \mathrm{N}, \exists a : \alpha, F\,n\,a$
- $\forall (n : \mathrm{N})(a\,b : \alpha), F\,n\,a \to F\,n\,b \to a = b$

Take $f\,n$ be $\epsilon a, F\,n\,a$.
Computation done by equational reasoning.

Recursion scheme

Prove the existence of a general recursor
$\forall g : \alpha, \forall h : \mathtt{N} \to \alpha \to \alpha.$
$\exists! f : \mathtt{N} \to \alpha, f\,0 = g \wedge \forall n. f\,(S\,n) = h\,n\,(f\,n)$
Define $F : \mathtt{N} \to \alpha \to o$ inductively :

$$\frac{}{F\,0\,g} \qquad \frac{F\,n\,a}{F\,(S\,n)\,(h\,n\,a)}$$

Prove by induction on n

- $\forall n : \mathtt{N}, \exists a : \alpha, F\,n\,a$
- $\forall (n : \mathtt{N})(a\,b : \alpha), F\,n\,a \to F\,n\,b \to a = b$

Take $f\,n$ be $\epsilon a, F\,n\,a$.
Computation done by equational reasoning.

Recursion scheme

Prove the existence of a general recursor
$\forall g : \alpha, \forall h : \mathbb{N} \to \alpha \to \alpha.$
$\exists! f : \mathbb{N} \to \alpha, f\,0 = g \wedge \forall n. f\,(S\,n) = h\,n\,(f\,n)$
Define $F : \mathbb{N} \to \alpha \to o$ inductively :

$$\frac{}{F\,0\,g} \qquad \frac{F\,n\,a}{F\,(S\,n)\,(h\,n\,a)}$$

Prove by induction on n

- $\forall n : \mathbb{N}, \exists a : \alpha, F\,n\,a$
- $\forall (n : \mathbb{N})(a\,b : \alpha), F\,n\,a \to F\,n\,b \to a = b$

Take $f\,n$ be $\epsilon a, F\,n\,a$.
Computation done by equational reasoning.

Encoding in the pure Calculus of Constructions

$\mathit{nat} \equiv \forall\alpha : \mathsf{Set}, (\alpha \to \alpha) \to \alpha \to \alpha$

- No proof of $0 \neq 1$, need axiom $\mathit{true} = \mathit{false}$
- Recursor for $\alpha : \mathsf{Set}$ but bad computational behavior (predecessor function).
- Complex construction for recursor with $\alpha := A \to \mathsf{Prop}$
- No proof of the induction principle (need restriction to $\{x : \mathit{nat} | Nx\}$).
- Extraction to Ocaml

Outline

Underlying PTS

- Sorts : Prop, Type_i, Set=Type_0
- Type_i:Type_{i+1}, Prop:Type_1.
- $\text{Type}_i \subseteq \text{Type}_{i+1}$, $\text{Prop} \subseteq \text{Type}_1$.
- Impredicativity

$$\frac{x : A \vdash B : \text{Prop}}{\forall x : A, B : \text{Prop}}$$

- Predicativity $s = \text{Prop}$ or $s = \text{Type}_i$

$$\frac{\vdash A : s \qquad x : A \vdash B : \text{Type}_i}{\forall x : A, B : \text{Type}_i}$$

Declaration

```
Inductive I1 pars : Ar1 := ...
  | c : ∀ (x1:A1)..(xn:An), I1 pars u1..up
  ...
with I2 pars : Ar2 := ...
with ...
```

Terminology

- *pars* parameters (same for all definitions)
- Ar_j arity
- u_i index
- $\forall (x_1 : A_1)..(x_n : A_n), I_1\ pars\ u_1..u_p$ type of constructor
- A_i type of argument of constructor

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- Encoding in HOL
- Rules for COQ inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Typing condition

- Arities are of the form $\forall(y_1 : B_1)..(y_p : B_p), s$
  s is the sort of the inductive definition.
- Type of constructors C are well-typed:

$$(I_1 : \forall pars, Ar_1)..(I_k : \forall pars, Ar_k)\,(pars) \vdash C : s$$

  - if s is predicative (not Prop) then type of arguments of constructors are in the same universe:
    forall i, $A_i : s$ or $A_i :$ Prop
  - if s is Prop, we distinguish
    - predicative definitions $A_i :$ Prop
    - impredicative definitions (at least one i such that $A_i :$ Type)

Positivity condition

In Coq occurrences of I_j should occur strictly positively in types of arguments of constructors A_i:

- does not occur: $I_j \notin A_i$
- simple case $A_i = I_j\, t_1 \ldots t_p$
  (not necessarily the same parameters, $I_j \notin t_k$)
- functional case $A_i = \forall z : B_1, B_2$
  with $I_l \notin B_1$ and I_j strictly positive in B_2
- imbricated case : $A_i = J\, t_1 \ldots t_p$
  with J another inductive definition with parameters $X_1 \ldots X_r$.
  When $t_1 \ldots t_r$ are substituted for $X_1 \ldots X_r$ in the types of constructors of J, the strict positivity condition is satisfied.

Example of imbricated definition

Trees with arbitrary (finite) branching.

```
Inductive list A : Type
   := nil | cons : A → list A → list A.
Inductive tree A : Type
   := node : A → list (tree A) → tree A.
```

Equivalent to a mutually inductive definition

```
Inductive tree A : Type
   := node : A → forest A → tree A
with forest A : Type
   := empty
   | add : tree A → forest A → forest A.
```

Exercise

```
Inductive X : Set := intro : unit + X -> X.

Inductive dec (A : Prop) : Prop :=
    yes : A -> dec A | no : ~ A → dec A.
Inductive X : Prop := intro : dec X -> X.

Inductive option (A : Set) : Set :=
    None | Some : A -> option A.
Inductive X : Set → Set :=
    abs : ∀ (A:Set), X (option A) → X A
  | var : ∀ (A:Set), A → X A.
Inductive X (A : Set) : Set :=
    abs : X (option A) → X A
  | var : A → X A.
```

Exercise

```
Inductive X : Set := intro : unit + X -> X.

Inductive dec (A : Prop) : Prop :=
    yes : A -> dec A | no : ~ A → dec A.
Inductive X : Prop := intro : dec X -> X.

Inductive option (A : Set) : Set :=
    None | Some : A -> option A.
Inductive X : Set → Set :=
    abs : ∀ (A:Set), X (option A) → X A
  | var : ∀ (A:Set), A → X A.
Inductive X (A : Set) : Set :=
    abs : X (option A) → X A
  | var : A → X A.
```

Exercise

```
Inductive X : Set := intro : unit + X -> X.

Inductive dec (A : Prop) : Prop :=
    yes : A -> dec A | no : ~ A → dec A.
Inductive X : Prop := intro : dec X -> X.

Inductive option (A : Set) : Set :=
    None | Some : A -> option A.
Inductive X : Set → Set :=
    abs : ∀ (A:Set), X (option A) → X A
  | var : ∀ (A:Set), A → X A.
Inductive X (A : Set) : Set :=
    abs : X (option A) → X A
  | var : A → X A.
```

Exercise

```
Inductive X : Set := intro : unit + X -> X.

Inductive dec (A : Prop) : Prop :=
    yes : A -> dec A | no : ~ A → dec A.
Inductive X : Prop := intro : dec X -> X.

Inductive option (A : Set) : Set :=
    None | Some : A -> option A.
Inductive X : Set → Set :=
    abs : ∀ (A:Set), X (option A) → X A
  | var : ∀ (A:Set), A → X A.
Inductive X (A : Set) : Set :=
    abs : X (option A) → X A
  | var : A → X A.
```

Outline

- Introduction
- Inductive constructions in practice
  - Basic data types
  - Predicate definition
  - Inductive families
  - Recursive functions
- Encoding in HOL
- Rules for Coq inductive constructions
  - Well-formedness of definition
  - Introduction
  - Elimination

Introduction rules

Given by the constructors.
c is the i-th constructor of inductive definition I with parameters $pars$ and type of constructor C.

$$c \equiv \texttt{Constr}(i, I) : \forall pars, C$$

Outline

Case analysis

- Induction principle versus Case analysis + fixpoint (cf Th. Coquand)
- (Primitive) pattern-matching is simple (one level, complete)
- Parameters are instantiated

$$\frac{\begin{array}{l} t : I\,pars\,t_1..t_p \\ y_1..y_k, x : I\,pars\,y_1..y_k \vdash P(y_1..y_k, x) : s' \\ (x_1 : A_1..x_n : A_n \vdash f : P(u_1..u_k, c\,x_1..x_n))_c \end{array}}{\begin{array}{l} \textbf{match}\ t\ \textbf{as}\ x\ \textbf{in}\ I\ \_\ y_1..y_k\ \textbf{return}\ P(y_1..y_k, x) \\ \textbf{with} \ldots | c\,x_1..x_n \Rightarrow f | \ldots \\ \textbf{end} : P(t_1..t_p\,t) \end{array}}$$

Reduction rules (ι) as expected when t starts with a constructor.

Inductive definitions and sorts

Which sort s' when doing case analysis on I of sort s?

- if s is Type, predicative inductive definition, any possible sort for case analysis.
- if s is Prop, impredicative sort + proof irrelevance interpretation + extraction
  - General case: only sort Prop for elimination. Strong elimination : $F\,(x : I) : Set$.
  - Particular cases : I is a predicative definition with only zero or one constructor (all A_i : Prop) any possible sort for case analysis.
    - absurdity (no constructor)
    - equality (no arguments)
    - conjunction of propositions
    - corresponds to Harrop's formula

Inductive definitions and sorts

Which sort s' when doing case analysis on I of sort s?

- if s is Type, predicative inductive definition,
  any possible sort for case analysis.
- if s is Prop, impredicative sort + proof irrelevance interpretation + extraction
  - General case: only sort Prop for elimination. Strong elimination : $F\,(x : I) : Set$.
  - Particular cases : I is a predicative definition with only zero or one constructor (all A_i : Prop)
    any possible sort for case analysis.
    - absurdity (no constructor)
    - equality (no arguments)
    - conjunction of propositions
    - corresponds to Harrop's formula

Inductive definitions and sorts

Which sort s' when doing case analysis on I of sort s?

- if s is Type, predicative inductive definition,
  any possible sort for case analysis.
- if s is Prop, impredicative sort + proof irrelevance interpretation + extraction
  - General case: only sort Prop for elimination. Strong elimination : $F\,(x : I) : Set$.
  - Particular cases : I is a predicative definition with only zero or one constructor (all A_i : Prop)
    any possible sort for case analysis.
    - absurdity (no constructor)
    - equality (no arguments)
    - conjunction of propositions
    - corresponds to Harrop's formula

Examples

```
Inductive sig (A:s1) (B:A→s2) : s :=
   pair : ∀ x:A,B x → sig A B.
Def fst (p : sig A B) : A :=
  match p return A with pair a b ⇒ a end.
Def snd (p : sig A B) : B (fst p) :=
  match p return B (fst p)
  with pair a b ⇒ b end.
```

- What are the possible relations between s_1, s_2, s ?
- In which cases can we define `fst` and `snd` ?

Records

Syntactic sugar for definition of tuples
automatic generation of projections

```
Record divspec (a b : nat) : Set := mkdiv
  {quo : nat; rem : nat;
   prop1 : a=b*quo+rem; prop2:rem<b
  }

Record monoid : Type := mkgrp
  {car:Type; op:car → car → car; elt : car;
   assoc : ∀ x y z, op (op x y) z = op x (op y z);
   neutr1 : ∀ x, op elt x = x;
   neutr2 : ∀ x, op x elt = x
   }
```

Exercice

Which elimination is used to prove $true \neq false$?

```
Inductive or (A B:Prop) : Prop :=
   left : A → or A B | right : B → or A B.
```

Prove $p\,q : or\ True\ True, p \neq q$ by case analysis on Set.

Recursive definition

- Concrete declaration:

  Fixpoint f $(x_1 : A_1) \ldots (x_m : A_m)$ {struct x_n}:B:=t.

- Internal fixpoint construction

  fix f $(x_1 : A_1) \ldots (x_n : A_n) : \forall (x_{n+1} : A_{n+1})$ $(x_m : A_m)$ B
  :=**fun** $x_{n+1} .. x_m$ ⇒ t.

- Typing condition:

$$(f : \forall (x_1 : A_1) \ldots (x_n : A_n), B)(x_1 : A_1) \ldots (x_n : A_n) \vdash t : B$$

- Condition: Recursive calls to f in t should be made on terms structurally smaller than x_n

Guarded definitions

Syntactic criteria: t is structurally smaller than x_n if

- $t = x\,\vec{u}$ with x a variable in a pattern in a **match** on x_n corresponding to a recursive argument.

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

- transitivity:

```
 Fixpoint div2 {struct n} : nat :=
  match n with
     O ⇒ O
   | (S p) ⇒ match p with
                     O ⇒ O | (S q) ⇒ S (div2 q)
                end.
  end.
```

Guarded definitions

Match construction: **match** u **with** $p_1 \Rightarrow u_1 | \ldots$ **end** is structurally smaller than x when each branch u_i is.

- informal explanation:
  f (**match** u **with** $p_1 \Rightarrow u_1 | \ldots$ **end**)
  is computationally equivalent to: **match** u **with** $p_1 \rightarrow f\ u_1 \ldots$ **end**
- When $u : False$, **match** u **with end** is structurally smaller than any term.
- ```
  Def pred n : 0<n → nat :=
    match n with O ⇒ fun H ⇒ error?False
               | S p ⇒ fun H ⇒ p end
  ```

  pred n H is smaller than n.
  ```

Well-founded recursion

Define *acc* as a relation in Prop.

```
Inductive acc (x:A) : Prop
   := acci : (∀ y, y<x → acc y) → acc x.
```

Given F of type $\forall x, (\forall y, y < x \to P\, y) \to P\, x$

```
Fixpoint wf_rec (x:A)(p:acc x) {struct p} : P x :=
  F x (fun y (h:y<x) ⇒
       wf_rec y (match p with (acci t) ⇒ t y h end)
```

- Works for $P\, x$: Type
- Does not use p for computation in the extracted term

A similar (more involved) trick can be used for the evaluation function in the WHILE language.

Computation

- Naive fixpoint reduction breaks strong normalisation
- Trick : guard reduction by asking the inductive arguments to start with a constructor.

Induction principles

They can be obtained combining fixpoint and pattern-matching

- Automatically generated when introducing an inductive definition
- Dependent version in general, non-dependent version for inductive in Prop.

```
∀ n, ∀ P,
 P n → (∀ m, le n m → P m → P (S m))
 → ∀ m, le n m → P m.
```

- Dependent or mutual induction obtained with **Scheme** command:

```
∀ n, ∀ (P:∀ m, le n m → ⋆),
   P n leb
 → (∀ m (p:le n m), P m p → P (S m) (leS p))
 → ∀ m (p:le n m), P m p.
```

Part II

Inductive Constructions : advanced notions

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Convertibility

Convertibility modulo $\beta\delta\iota\ldots$.

- Meta-theoretical notion corresponding to the same λ-term
- Two convertible propositions have the same proofs

$$(2 + 2 > 2) \equiv (4 > 2)$$

- Intensional equality ($\neq$ extensional): two different algorithms for sorting are not convertible

Inductive equality

Leibniz equality, smallest reflexive relation

- Polymorphic binary predicate $\forall\alpha, \alpha \to \alpha \to \text{Prop}$
- Strong link with convertibility:

$$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash \mathit{refleq} : t = u}$$

if $p : t = u$ in the empty context then $t \equiv u$ (meta-theorem)
Proofs of $\forall n, 0 + n = n \qquad \forall n, n + 0 = n$

Leibniz equality and dependent types

- How to compare objects in different types?
  - ```
    Inductive list :=
       nil : list O
    | cons : ∀ n, A → list n → list (S n).
    Def app : ∀ n m, list n → list m → list (n+m).
    Lemma ∀ n (l:list n), app nil l = l.

    Lemma ∀ n (l:list n), app l nil = l.
       NOT WELL TYPED!
    ```
- Idea: compare $(n+0, app\ l\ nil) = (n, l)$ in $\Sigma n, list\ n$
  But no possible replacement
    ```

Leibniz equality and dependent types

- How to compare objects in different types?
  - ```
    Inductive list :=
        nil : list O
      | cons : ∀ n, A → list n → list (S n).
    Def app : ∀ n m, list n → list m → list (n+m).
    Lemma ∀ n (l:list n), app nil l = l.
    ```
    ```
    Lemma ∀ n (l:list n), app l nil = l.
        NOT WELL TYPED!
    ```
- Idea: compare $(n+0, app\, l\, nil) = (n, l)$ in $\Sigma n, list\, n$
  But no possible replacement
    ```

Equality on dependent types

$(A : \mathrm{Type})\,(P : A \to \mathrm{Type})\,(a\,b : A)\,(t : P\,a)\,(u : P\,b)$
How to say that $t = u$?

```
Def eqdep (a b:A) (t:P a) (u:P b) := (a,t)=(b,u).
Inductive eqdep (a:A) (t:P a) : ∀ b, P b → Prop :=
    refleqdep : eqdep a t a t.
Def eqdep (a b:A) (t:P a) (u:P b)
   := exists h:a=b, subst h t=u.
```

- Equivalent relations
- None of them can prove : $\forall a (t\,u : P\,a), eqdep\,t\,u \to t = u$
- Related to the absence of proof of : $\forall x\,(p : x = x), p = refleq\,x$.

Context of substitution

Elimination principle for inductive equality:

```
∀ (P:∀ y, x=y → ⋆), P x (refleq x)
   → ∀ y (p:x=y), P y p
```

Only says that $(y, p) = (x, \mathit{refleqx})$
Property $Q\,p \equiv p = \mathit{refleq}\,x$ only well-typed if $p : x = x$ cannot be abstracted such that $p : x = y$.

- problem first identified by Th. Coquand
- models where it is not true (M. Hoffman, Th Streicher)

Dependent equality in practice

- Problem appears even in simple examples:

  `∀ l:list O, l=nil.`

  cannot directly use case analysis on *l* which requires to abstract with respect to *n* and $l : list\ n$.
- K axiom (Streicher's habilitation): equivalent to $\forall x(p : x = x), p = refleq\ x$.
  In Coq (file `Logic/Eqdep`):
  $\forall U(p : U)(Q : U \rightarrow Type)(x : Q\ p)(h : p = p),$
  $x = \textbf{match}\ h\ \textbf{with}\ refleq \Rightarrow x\ \textbf{end}$
- Axiom provable when equality on *U* is decidable.
  M. Hedberg, Th. Kleymann (Lego), B. Barras (Coq `Eqdep_dec`)

Dependent equality in practice

- Problem appears even in simple examples:

  ```
  ∀ l:list O, l=nil.
  ```

  cannot directly use case analysis on l which requires to abstract with respect to n and $l : list\ n$.
- K axiom (Streicher's habilitation): equivalent to $\forall x (p : x = x), p = refleq\ x$.
  In Coq (file `Logic/Eqdep`):
  $\forall U (p : U)(Q : U \to Type)(x : Q\ p)(h : p = p),$
  $x =$ **match** h **with** $refleq \Rightarrow x$ **end**
- Axiom provable when equality on U is decidable.
  M. Hedberg, Th. Kleymann (Lego), B. Barras (Coq `Eqdep_dec`)

Dependent equality in practice

- Problem appears even in simple examples:

  `∀ l:list O, l=nil.`

  cannot directly use case analysis on l which requires to abstract with respect to n and $l : list\ n$.
- K axiom (Streicher's habilitation): equivalent to
  $\forall x (p : x = x), p = refleq\ x$.
  In Coq (file `Logic/Eqdep`):
  $\forall U (p : U)(Q : U \rightarrow Type)(x : Q\ p)(h : p = p),$
  $x = \textbf{match}\ h\ \textbf{with}\ refleq \Rightarrow x\ \textbf{end}$
- Axiom provable when equality on U is decidable.
  M. Hedberg, Th. Kleymann (Lego), B. Barras (Coq `Eqdep_dec`)

Dependent equality on nat

```
Def eqdnat (n m : nat) : ∀ P, P n → P m → Prop
  eqdnat 0 0 P t u ⇒ t=u
  eqdnat (S p) (S q) P t u ⇒ eqdnat p q (P o S) t u
  eqdnat _ _ P t u ⇒ False
```

It is easy to prove the following facts by induction on *n*.

```
∀ n P (p:P n), eqdnat n n P p p
∀ n P (p q:P n), eqdnat n n P p q → p=q
```

We deduce

```
∀ n m P (p:P n)(q:P m), eqdep p q → eqdnat n m P p q
∀ n P (p q:P n), eqdep p q → p=q
```

Heterogeneous equality

Previously called John Major's equality (Conor McBride).
Compare $x : A$ with $y : B$ with arbitrary A, B
True when A and B are convertible, as well as x and y.

```
Inductive Heq (A:Type) (x:A) : ∀ B, B → Prop :=
    reflHeq : Heq A x A x.
```

Symmetry and transitivity can be proved as for Leibniz equality.
Proof of $Heq\,A\,x\,B\,y \rightarrow A = B$.
More or less useless without an axiom (equivalent to K).

```
∀ A (x y:A), Heq A x A y → x=y.
```

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$\begin{array}{lll} \text{omega} & \equiv & \text{app (Lam (}\textbf{fun}\text{ x} \Rightarrow \text{app x x)) delta} \\ & \longrightarrow & (\textbf{fun}\text{ x} \Rightarrow \text{app x x) delta} \\ & \longrightarrow & \text{app delta delta} \equiv \text{omega} \end{array}$$

Not compatible with higher-order syntax for binders representation . . .

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$\begin{array}{lll} \text{omega} & \equiv & \text{app (Lam (}\textbf{fun}\text{ x} \Rightarrow \text{app x x)) delta} \\ & \longrightarrow & \text{(}\textbf{fun}\text{ x} \Rightarrow \text{app x x) delta} \\ & \longrightarrow & \text{app delta delta} \equiv \text{omega} \end{array}$$

Not compatible with higher-order syntax for binders representation . . .

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$\begin{array}{lll} \text{omega} & \equiv & \text{app (Lam (fun } x \Rightarrow \text{app } x\ x)) \text{ delta} \\ & \longrightarrow & (\text{fun } x \Rightarrow \text{app } x\ x) \text{ delta} \\ & \longrightarrow & \text{app delta delta} \equiv \text{omega} \end{array}$$

Not compatible with higher-order syntax for binders representation ...

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$\begin{array}{lll} \text{omega} & \equiv & \text{app (Lam (fun } x \Rightarrow \text{app } x\ x))\ \text{delta} \\ & \longrightarrow & (\text{fun } x \Rightarrow \text{app } x\ x)\ \text{delta} \\ & \longrightarrow & \text{app delta delta} \equiv \text{omega} \end{array}$$

Not compatible with higher-order syntax for binders representation …

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$
\begin{array}{lll}
\text{omega} & \equiv & \text{app (Lam (fun } x \Rightarrow \text{app } x\ x)) \text{ delta} \\
 & \longrightarrow & (\text{fun } x \Rightarrow \text{app } x\ x) \text{ delta} \\
 & \longrightarrow & \text{app delta delta} \equiv \text{omega}
\end{array}
$$

Not compatible with higher-order syntax for binders representation . . .

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

$$\begin{array}{lll} \text{omega} & \equiv & \text{app (Lam (}\mathbf{fun}\ \text{x} \Rightarrow \text{app x x)) delta} \\ & \longrightarrow & (\mathbf{fun}\ \text{x} \Rightarrow \text{app x x) delta} \\ & \longrightarrow & \text{app delta delta} \equiv \text{omega} \end{array}$$

Not compatible with higher-order syntax for binders representation . . .

Positivity

A negative occurrence in a type of constructor gives non terminating terms even without recursion.

```
Inductive L = Lam : (L → L) → L.

Def app : L → L → L :=
    app (Lam f) x ⇒ f x.

Def delta : L := Lam (fun x ⇒ app x x)
Def omega : L := app delta delta
```

omega	$\equiv$	app (Lam (**fun** x $\Rightarrow$ app x x)) delta
	$\longrightarrow$	(**fun** x $\Rightarrow$ app x x) delta
	$\longrightarrow$	app delta delta $\equiv$ omega

Not compatible with higher-order syntax for binders representation . . .

General positivity

Strict positivity required at the Type level:

```
Inductive B : Type := in : ((B → Prop) → Prop) → B
```

```
Def f (P:B→ Prop) : Prop := in (fun Q ⇒ P=Q)
Lemma ∀ P Q, f P = f Q → P = Q
```

Paradox: $P_0\,x := \exists P.f\,P = x \wedge \neg P\,x \quad i_0 := f\,P_0$

$$P_0\,i_0 \Leftrightarrow \neg P_0\,i_0$$

Monotonicity is correct at the impredicative level Prop (but not implemented).

```
Def orc A B :=
  ∀ C, (¬ ¬ C → C) → (A → C) → (B → C) → C
```

General positivity

Strict positivity required at the Type level:

```
Inductive B : Type := in : ((B → Prop) → Prop) → B
```

```
Def f (P:B→ Prop) : Prop := in (fun Q ⇒ P=Q)
Lemma ∀ P Q, f P = f Q → P = Q
```

Paradox: $P_0\,x := \exists P.f\,P = x \wedge \neg P\,x \quad i_0 := f\,P_0$

$$P_0\,i_0 \Leftrightarrow \neg P_0\,i_0$$

Monotonicity is correct at the impredicative level Prop
(but not implemented).

```
Def orc A B :=
  ∀ C, (¬ ¬ C → C) → (A → C) → (B → C) → C
```

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Impredicative type

```
Inductive A : Prop := in : Prop → A.
Def out : A → Prop :=
    out (in P) ⇒ P.
```

We end up with A : Prop and $A \leftrightarrow$ Prop which gives a paradox.
Not allowed in COQ because it requires case analysis on predicate $P\ x :=$ Prop : Type

Classical logic

Implies proof-irrelevance.

```
Inductive BOOL : Prop := T | F.
Lemma (∀ A:Prop, A ∨ ¬ A) → T=F.
```

Cf Coq-lab : $I : \mathrm{Prop} \rightarrow BOOL$ such that $(I\,A = T) \leftrightarrow A$.

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Indecidability of completeness of pattern-matching

Nicolas Oury
Post-problem : pairs of words (u_1, v_1),ldots (u_n, v_n)
Search solutions $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$

```
Inductive char := A | B. Def word := list char.
Inductive post : word → word → Prop :=
  post0 : post nil nil
 |post1 : ∀ l m, post l m → post u1[l] v1[m]
...
 |postn : ∀ l m, post l m → post un[l] vn[m]
```

Purely first-order inductive definition.

```
Def nondec : ∀ l, post l l → unit :=
     nondec nil post0 ⇒ tt.
```

Complete definition: no non-trivial solution to the post-problem.

Guard condition

Recursion only on recursive arguments

```
Inductive I (A:Prop) : Prop := c : A → I A.
Def Tr : Prop := ∀ A, A → A.
Def id : True := fun A x → x.

Def f (I Tr) → X :=
     f (c u) ⇒ f (u (I True) (c id)).
f (c id) ⟶ f (c id)

Def f (I True) → X :=
     f (c u) ⇒ f (match u with tt ⇒ c tt)).
f (c tt) ⟶ f (c tt)
```

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Introduction

- Coinductive definitions are greatest fixpoints of monotonic operators
- $C = \nu X.F\,X$ satisfies

$$F\,C \subseteq C \qquad C \subseteq F\,C \qquad \forall X, (X \subseteq F\,X) \rightarrow X \subseteq C$$

- Impredicative encoding :

```
Def C x := ∃ X, (X⊆ F X) ∧ (X x)
```

  Abstract type X, a state $s : X\,x$ and a method $f : X \subseteq F\,X$ to produce outputs and new states.

Example of streams

Infinite lists (streams)

$F\,X := A * X$

$S := \exists X, X \to A * X \wedge X$ (record).

- Coiterative construction of a stream : $< X, f, s >$ with $f : X \to A * X$ and $s : X$

```
Def Coit X (f:X → A * X) (s:X) : S := <X,f,s>.
```

- Head/Tail fonctions :

```
Def hd : S → A :=
    hd <X,f,s> ⇒ f s.
Def tl : S → S :=
    tl <X,f,s> ⇒ <X,f,f s>.
```

- Properties : $hd\,(CoIt\,f\,s) = fst\,(f\,s)$
  $tl\,(CoIt\,f\,s) = (CoIt\,f\,(snd\,(f\,s))$

Constructor

cons a s first outputs *a* then behaves like *s*.
Need to distinguish the first step

```
Def cons : A*S → S :=
  cons (a, <X,f,s>) ⇒ <option X, g a f s, None>.

Def g (a:A) X (f:X → A * X) (s:X)
     : option X → A * option X :=
 g a X f s None ⇒ (a, Some s)
 g a X f s (Some y) ⇒ let (b,z) := f y in (b, Some z)
```

Alternative (more abstract) definition :

```
Def cons (a, s) ⇒ <option S, g a s, None>.
Def g (a:A) (s:S) : option S → A * option S :=
 g a s None ⇒ (a, Some s)
 g a s (Some x) ⇒ hd x, Some (tl x)
```

Constructor

cons a s first outputs *a* then behaves like *s*.
Need to distinguish the first step

```
Def cons : A*S → S :=
   cons (a, <X,f,s>) ⇒ <option X, g a f s, None>.

Def g (a:A) X (f:X → A * X) (s:X)
      : option X → A * option X :=
  g a X f s None ⇒ (a, Some s)
  g a X f s (Some y) ⇒ let (b,z) := f y in (b, Some z)
```

Alternative (more abstract) definition :

```
Def cons (a, s) ⇒ <option S, g a s, None>.
Def g (a:A) (s:S) : option S → A * option S :=
  g a s None ⇒ (a, Some s)
  g a s (Some x) ⇒ hd x, Some (tl x)
```

Constructor

cons a s first outputs *a* then behaves like *s*.
Need to distinguish the first step

```
Def cons : A*S → S :=
  cons (a, <X,f,s>) ⇒ <option X, g a f s, None>.

Def g (a:A) X (f:X → A * X) (s:X)
     : option X → A * option X :=
 g a X f s None ⇒ (a, Some s)
 g a X f s (Some y) ⇒ let (b,z) := f y in (b, Some z)
```

Alternative (more abstract) definition :

```
Def cons (a, s) ⇒ <option S, g a s, None>.
Def g (a:A) (s:S) : option S → A * option S :=
 g a s None ⇒ (a, Some s)
 g a s (Some x) ⇒ hd x, Some (tl x)
```

Functional programming point of view

Th. Coquand

- Concrete data structure like inductive definition but with possible infinite elements.
- Case analysis but no induction principle
- Fixpoint definition of infinite objects but with a guard condition for productivity.

Example of streams

```
CoInductive Str (A:Set) : Set :=
    cons : A → Str A → Str A.
```

Projections

```
Def hd A (s:Str A) : A :=
     hd A (cons a s) ⇒ a.
Def tl A (s:Str A) : Str A :=
     hd A (cons a s) ⇒ s.
```

Recursive definition

```
Def cte A (a:A) : Str A := cons a (cte A a).
Def CoIt X (f:X→ A*X) (s:X) : Str A
   := cons (fst (f s)) (CoIt X s (snd (f s))).
```

Exercise : define the map function

Example of streams

```
CoInductive Str (A:Set) : Set :=
   cons : A → Str A → Str A.
```

Projections

```
Def hd A (s:Str A) : A :=
     hd A (cons a s) ⇒ a.
Def tl A (s:Str A) : Str A :=
     hd A (cons a s) ⇒ s.
```

Recursive definition

```
Def cte A (a:A) : Str A := cons a (cte A a).
Def CoIt X (f:X→ A*X) (s:X) : Str A
   := cons (fst (f s)) (CoIt X s (snd (f s))).
```

Exercise : define the map function

Example of streams

```
CoInductive Str (A:Set) : Set :=
   cons : A → Str A → Str A.
```

Projections

```
Def hd A (s:Str A) : A :=
     hd A (cons a s) ⇒ a.
Def tl A (s:Str A) : Str A :=
     hd A (cons a s) ⇒ s.
```

Recursive definition

```
Def cte A (a:A) : Str A := cons a (cte A a).
Def CoIt X (f:X→ A*X) (s:X) : Str A
   := cons (fst (f s)) (CoIt X s (snd (f s))).
```

Exercise : define the map function

Productivity condition

Limit of recursive definitions of streams:

```
Def filter A (p:A→ bool) (s:Str A) : Str A :=
  filter A p (cons a t) ⇒
     if p a then cons a (filter p t) else filter p t
```

Problem

```
match filter p s with cons a _ ⇒ ... end
```

Productivity condition : the recursive call appears immediatly under a constructor.

Productivity condition

Limit of recursive definitions of streams:

```
Def filter A (p:A→ bool) (s:Str A) : Str A :=
  filter A p (cons a t) ⇒
     if p a then cons a (filter p t) else filter p t
```

Problem

```
match filter p s with cons a _ ⇒ ... end
```

Productivity condition : the recursive call appears immediatly under a constructor.

Reduction rules

cofix $f\,x := t$

- A co-fixpoint is a normal form.
- fixpoint reduction when fixpoint is in a **match** operation.

$$\textbf{match}\ f\,x\ \textbf{with}\ p \Rightarrow \ldots\ \textbf{end} \longrightarrow \textbf{match}\ t\ \textbf{with}\ p \Rightarrow \ldots\ \textbf{end}$$

- systematic proof of $f\,x = t$ with Leibniz equality using

$$s = \textbf{match}\ s\ \textbf{with}\ (cons\,a\,u) \Rightarrow cons\,a\,u\ \textbf{end}$$

Equality on streams

- Intensional equality is not appropriate: canonical streams are generated by different algorithms.

```
Def alt1 : Str bool := cons true (cons false alt1).
Def alt2 (b:bool): Str bool
  := cons b (alt2 (not b)).
```

- Two streams are equal if they have the same elements

```
eqStr x y ↔ hd x = hd y ∧ eqStr (tl x) (tl y)
```

- *eqStr* should be defined co-inductively.
  Proofs of equality are also co-recursively defined.

```
Def alt1_2 : eqStr alt1 (alt2 true) :=
   eqStr_i alt1 (alt2 true) (refleq true)
          (eqStr_i (cons false alt1) (alt2 false)
                  (refleq false) alt1_2.
```

Combining induction and co-induction

- Define a function `pre` of type $Str\ A \rightarrow nat \rightarrow list\ A$ such that $pre\ s\ n$ contains the first n elements of s.
  - Recursion on n.
- Show that $\forall st, eqStr\ s\ t \rightarrow \forall n, pre\ s\ n = pre\ t\ n$
  - Induction on n
- Show the opposite direction.
  - Co recursion
- Given a property P on A, define a property on streams which says that P is true:
  - for all the elements of the stream
  - for at least one element in the stream
  - for infinitely many elements in the stream

Combining induction and co-induction

- Define a function `pre` of type $Str\ A \rightarrow nat \rightarrow list\ A$ such that $pre\ s\ n$ contains the first n elements of s.
  - Recursion on n.
- Show that $\forall st, eqStr\ s\ t \rightarrow \forall n, pre\ s\ n = pre\ t\ n$
  - Induction on n
- Show the opposite direction.
  - Co recursion
- Given a property P on A, define a property on streams which says that P is true:
  - for all the elements of the stream
  - for at least one element in the stream
  - for infinitely many elements in the stream

Combining induction and co-induction

- Define a function `pre` of type $Str\ A \rightarrow nat \rightarrow list\ A$ such that $pre\ s\ n$ contains the first n elements of s.
  - Recursion on n.
- Show that $\forall st, eqStr\ s\ t \rightarrow \forall n, pre\ s\ n = pre\ t\ n$
  - Induction on n
- Show the opposite direction.
  - Co recursion
- Given a property P on A, define a property on streams which says that P is true:
  - for all the elements of the stream
  - for at least one element in the stream
  - for infinitely many elements in the stream

Combining induction and co-induction

- Define a function `pre` of type $Str\ A \rightarrow nat \rightarrow list\ A$ such that $pre\ s\ n$ contains the first n elements of s.
  - Recursion on n.
- Show that $\forall st, eqStr\ s\ t \rightarrow \forall n, pre\ s\ n = pre\ t\ n$
  - Induction on n
- Show the opposite direction.
  - Co recursion
- Given a property P on A, define a property on streams which says that P is true:
  - for all the elements of the stream
  - for at least one element in the stream
  - for infinitely many elements in the stream

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Outline

Induction recursion

- Used in the definition of universes in MLTT, studied by P. Dybjer. Introduce a type U of codes of propositions with a decoding function of type $U \rightarrow \mathsf{Set}$.
  We want
  - $\dot{nat} : U$ with $dec\, \dot{nat} \Rightarrow nat$
  - $\dot{+} : U \rightarrow U \rightarrow U$ with $dec\, \dot{+}\, x\, y \Rightarrow dec\, x + dec\, y$
  - $dec\,(\dot{\Pi}\, A\, B) = \forall x : dec\, A, dec\,(B\, x)$ dec appears in type of $\dot{\Pi}$.

```
Inductive U : Type :=
    cnat : U
    csum : U -> U -> U
    cpi  : ∀ (A:U) (dec A → U) → U
with dec : U → Set :=
    dec cnat ⇒ nat
    dec (csum A B) ⇒ csum A + csum B
    dec (cpi A B) ⇒ ∀ x:dec A, dec (B x)
```

Encoding in Coq

```
Inductive UT : Set → Type :=
   UTnat : UT nat
 | UTsum : ∀ A B, UT A → UT B → UT (A+B)
 | UTpi : ∀ (A:Set) (B:A→ Set), UT A →
           (∀ x:A, UT (B x)) → UT (∀ x:A, B x).
Record U : Type := mkU {dec:Set; val:UT dec}.
Def cnat : U := mkU UTnat.
Def csum (x y: U) : U :=
    mkU (UTsum (val x) (val y)).
Def cpi (A : U) (B : dec A → U) : U :=
  mkU (UTpi (fun z ⇒ dec (B z))
            (val A) (fun z ⇒ val (B z))).
```

$dec\,(cpi\,A\,B) \equiv \forall x : dec\,A, dec\,(B\,x)$

Outline

- Equality
- Paradoxes
  - Positivity condition
  - Sorts
  - Guarded definitions and pattern-matching
- Coinductive definitions
- Extensions
  - Induction-recursion
  - Size-annotation
  - Algebraic constructions

Size annotation

- Guard condition for fixpoints is a global side condition
  - syntactic criteria
  - interact badly with reduction, tactics
  - not powerful enough for imbricated recursive definition
- Use instead a typing relation with special marks Mendler, Giménez, Barthe, Amadio, Altenkirch ...

$$0 : nat^{\hat{x}} \quad S : nat^{x} \Rightarrow nat^{\hat{x}}$$

$$\frac{n : \hat{x} \quad g : P\,0 \quad n : nat^{x} \vdash h : P\,(S\,n)}{\textbf{match}\ n\ \textbf{with}\ 0 \Rightarrow g | S\,n \Rightarrow h\ \textbf{end} : P\,n}$$

$$\frac{f : nat^{x} \rightarrow \alpha \vdash t : nat^{\hat{x}} \rightarrow \alpha}{\textbf{fix}\ f\ x := t : nat^{\infty} \rightarrow \alpha}$$

Outline

Rewriting in conversion

Consider extensions of lambda-calculus with new types and reduction rules.

- $n + 0 \longrightarrow n \quad 0 + n \longrightarrow n \quad (n + m) + q \longrightarrow n + (m + q)$
- Algebraic extensions of simple lambda-calculus (Breazu-Tannen, Fernandez, Barbanera ...)
- General scheme (Blanqui, Jouannaud, Okada ...)
- RPO (Walukiewicz, Jouannaud, Rubio)

Many questions:

- Normalisation,
- Confluence,
- Consistency,
- Completeness,
- Efficiency of reduction

Summary

Inductive definitions

- useful notion in computer science
- strong constructive interpretation
- powerful notion
  (models or encoding are useful to ensure consistency)
- interaction of programming and logic still problematic
  - termination
  - completeness of pattern-matching
  - interaction with proof-irrelevance
- better interfaces in proof assistants
  - termination criteria
  - induction principles

Bibliography

- Martin-Löf (P.). – Constructive mathematics and computer programming. *In : Logic, Methodology and Philosophy of Science*. pp. 153–175. – North-Holland.
- Bertot (Y.) and Castéran (P.). *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
- The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, Feb 2007. `http://coq.inria.fr`.
- A Tutorial on Recursive Types in Coq by E. Giménez and P. Castéran `http://www.labri.fr/Perso/~casteran/RecTutorial.pdf.gz`
- …