Programming with (co-)inductive types in Coq

Matthieu Sozeau

February 3rd 2014
Programming with (co-)inductive types in Coq

Matthieu Sozeau

February 3rd 2014
Last time

1. Record Types
2. Mathematical Structures and Coercions
3. Type Classes and Canonical Structures
4. Interfaces and Implementations
5. TODO Monadic Programming with Type Classes

Questions?
matthieu.sozeau@inria.fr
In this class, we shall present how Coq allows us in practice to define data types using (co-)inductive declarations, compute on these datatypes, and reason by induction.
Inductive declarations

An arbitrary type as assumed by:

Variable T : Type.

gives no a priori information on the nature, the number, or the properties of its inhabitants.
An *inductive* type declaration explains how the inhabitants of the type are built, by giving *names* to each construction rule:
An **inductive** type declaration explains how the inhabitants of the type are built, by giving **names** to each construction rule:

Print bool.
Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule:

```
Print bool.
```

Each such rule is called a constructor.
Inductive declarations

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule:

Print bool.

Print nat.
An **inductive** type declaration explains how the inhabitants of the type are built, by giving **names** to each construction rule:

Print bool.


Print nat.

An inductive type declaration explains how the inhabitants of the type are built, by giving names to each construction rule:

Print bool.


Print nat.


Each such rule is called a constructor.
Enumerated types

Enumerated types are types which list and name exhaustively their inhabitants.


Inductive color:Type :=
| white | black | yellow | cyan | magenta
| red | blue | green.

Check cyan.
cyan : color
Enumerated types

Enumerated types are types which list and name exhaustively their inhabitants.


Inductive color:Type :=
| white | black | yellow | cyan | magenta
| red | blue | green.

Check cyan.
* cyan : color

Labels refer to distinct elements.
Enumerated types: program by case analysis

Inspect the enumerated type inhabitants and assign values:

Definition my_negb (b : bool) :=
  match b with true => false | false => true.

Definition is_black_or_white (x : color) : bool :=
  match x with
  | black => true
  | white => true
  | _ => false
  end.

Compute: constructors are values.
Eval compute in (is_black_or_white hat).
= false : bool
Enumerated types: program by case analysis

Inspect the enumerated type inhabitants and assign values:

Definition my_negb (b : bool) :=
    match b with true => false | false => true.

Definition is_black_or_white (x : color) : bool :=
    match x with
    | black => true
    | white => true
    | _ => false
    end.
Enumerated types: program by case analysis

Inspect the enumerated type inhabitants and assign values:

Definition my_negb (b : bool) :=
    match b with true => false | false => true.

Definition is_black_or_white (x : color) : bool :=
    match x with
    | black => true
    | white => true
    | _ => false
    end.

Compute: constructors are values.

Eval compute in (is_black_or_white hat).
Enumerated types: program by case analysis

Inspect the enumerated type inhabitants and assign values:

Definition my_negb (b : bool) :=
    match b with true => false | false => true.

Definition is_black_or_white (x : color) : bool :=
    match x with
    | black => true
    | white => true
    | _ => false
    end.

Compute: constructors are values.

Eval compute in (is_black_or_white hat).

= false
: bool
Enumerated types: reason by case analysis

Inspect the enumerated type inhabitants and build proofs:

Lemma bool_case : forall b : bool, b = true \/ b = false.
Proof.
intro b.
case b.
  left; reflexivity.
right; reflexivity.
Qed.
Enumerated types: reason by case analysis

Inspect the enumerated type inhabitants and build proofs:

Lemma `is_black_or_whiteP : forall x : color, is_black_or_white x = true -> x = black \/ x = white.`

Proof.
(* Case analysis + computation *)
intro x; case x; simpl; intro e.
(* In the three first cases: e: false = true *)
discriminate e.
discriminate e.
discriminate e.
discriminate e.
(* Now: e: true = true *)
left; reflexivity.
right; reflexivity.
Qed.
Enumerated types: reason by case analysis

Two important tactics, not specific to enumerated types:

- **simpl**: makes computation progress (pattern matching applied to a term starting with a constructor)
- **discriminate**: allows to use the fact that constructors are distincts:
  - **discriminate H**: closes a goal featuring a hypothesis H like (H : true = false);
  - **discriminate**: closes a goal like (0 <> S n).
Options and 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: \texttt{option} \( B \).

\[
\text{Inductive option} : \text{Type} := \\
\quad \text{Some} : B \to \text{option} \mid \text{None} : \text{option}.
\]

- 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: \( \forall x, x \in D \to B \)
  - Argument erased by extraction: \( D : A \to \text{Prop} \).
  - Proof irrelevance: \( f x d_1 = f x d_2 \)
Recursive types

Let us craft new inductive types:

Inductive natBinTree : Set :=
  | Leaf : nat -> natBinTree

Inductive term : Set :=
  | Zero : term
  | One : term
  | Plus : term -> term -> term

An inhabitant of a recursive type is built from a finite number of constructor applications.
Recursive types

Let us craft new inductive types:

```
Inductive natBinTree : Set :=
  | Leaf : nat -> natBinTree
```

An inhabitant of a recursive type is built from a finite number of constructor applications.
Recursive types

Let us craft new inductive types:

Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
| Node : nat ->
Recursive types

Let us craft new inductive types:

Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
| Node : nat -> natBinTree -> natBinTree
Recursive types

Let us craft new inductive types:

Inductive natBinTree : Set :=
| Leaf : nat -> natBinTree
Recursive types

Let us craft new inductive types:

\[
\text{Inductive } \text{natBinTree} : \text{Set} := \\
\mid \text{Leaf} : \text{nat} \to \text{natBinTree} \\
\mid \text{Node} : \text{nat} \to \text{natBinTree} \to \text{natBinTree} \to \text{natBinTree}.
\]

\[
\text{Inductive } \text{term} : \text{Set} := \\
\mid \text{Zero} : \text{term} \\
\mid \text{One} : \text{term} \\
\mid \text{Plus} : \text{term} \to \text{term} \to \text{term} \\
\mid \text{Mult} : \text{term} \to \text{term} \to \text{term}.
\]

An inhabitant of a recursive type is built from a finite number of constructor applications.
Recursive types: program by case analysis

We have already seen some examples of such pattern matching:

Definition isNotTwo x :=
    match x with
        | S (S O) => false
        | _ => true
    end.

Definition is_single_nBT (t : natBinTree) :=
    match t with
        | Leaf _ => true
        | _ => false
    end.
Recursive types: program by case analysis

We have already seen some examples of such pattern matching:

Definition isNotTwo x :=
  match x with
  | S (S 0) => false
  | _ => true
end.

Definition is_single_nBT (t : natBinTree) :=
match t with
| Leaf _ => true
| _ => false
end.
Lemma is_single_nBTP : forall t,
is_single_nBTP t = true -> exists n : nat, t = Leaf n.
Proof.
Lemma is_single_nBTP : forall t, 
is_single_nBTP t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
   while introducing *)
intros [ nleaf | nnode t1 t2] h.
Lemma is_single_nBTP : forall t, 
  is_single_nBTP t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree 
   while introducing *)
intros [ nleaf | nnode t1 t2] h.
(* First case: we use the available label *)
exists nleaf.
reflexivity.
Lemma is_single_nBTP : forall t,
    is_single_nBTP t = true -> exists n : nat, t = Leaf n.
Proof.
(* We use the possibility to destruct the tree
   while introducing *)
intros [ nleaf | nnode t1 t2] h.
(* First case: we use the available label *)
    exists nleaf.
    reflexivity.
(* Second case: the test evaluates to false *)
simpl in h.
discriminate.
Lemma is_single_nBTP : forall t, 
    is_single_nBTP t = true -> exists n : nat, t = Leaf n. 
Proof. 
(* We use the possibility to destruct the tree 
    while introducing *) 
intros [ nleaf | nnode t1 t2] h. 
(* First case: we use the available label *) 
exists nleaf. 
  reflexivity. 
(* Second case: the test evaluates to false *) 
simpl in h. 
  discriminate. 
Qed.
Recursive types

Constructors are **injective**:

Lemma `inj_leaf` : for all `x` `y`, `Leaf x = Leaf y` \(\rightarrow\) `x = y`.  
Proof.  
intros `x` `y` `hLxLy`.  
injection `hLxLy`.  
trivial.  
Qed.
Recursive types: structural induction

Let us go back to the definition of natural numbers:

\[
\text{Inductive } \text{nat} : \text{Set} := \ 0 : \text{nat} \mid S : \text{nat} \to \text{nat}.\]
Recursive types: structural induction

Let us go back to the definition of natural numbers:

\[
\text{Inductive } \text{nat} : \text{Set} := \text{O} : \text{nat} \mid \text{S} : \text{nat} \rightarrow \text{nat}.
\]

The \textit{Inductive} keyword means that at definition time, this system generates an \textit{induction principle}:

\[
\text{nat\_ind} \\
: \text{forall } P : \text{nat} \rightarrow \text{Prop}, \\
P \text{ O } \rightarrow
\]

Let us go back to the definition of natural numbers:

```coq
```

The `Inductive` keyword means that at definition time, this system generates an induction principle:

```coq
nat_ind
  : forall P : nat -> Prop,
     P 0 ->
     (forall n : nat, P n -> P (S n)) ->
     forall n : nat, P n
```
Recursive types: structural induction

To prove that for $P : \text{term} \rightarrow \text{Prop}$, the theorem $\forall t : \text{term}, P t$ holds, it is sufficient to:

1. Prove that the property holds for the base cases: $\forall t : \text{term}, P \text{Zero}$, $\forall t : \text{term}, P \text{One}$
2. Prove that the property is transmitted inductively: $\forall t1 t2 : \text{term}, P t1 \rightarrow P t2 \rightarrow P (\text{Plus} t1 t2)$, $\forall t1 t2 : \text{term}, P t1 \rightarrow P t2 \rightarrow P (\text{Mult} t1 t2)$

The type $\text{term}$ is the smallest type containing $\text{Zero}$ and $\text{One}$, and closed under $\text{Plus}$ and $\text{Mult}$. 
Recursive types: structural induction

To prove that for $P : \text{term} \rightarrow \text{Prop}$, the theorem $\forall t : \text{term}, P \ t$ holds, it is sufficient to:

- Prove that the property holds for the base cases:
  - $(P \ \text{Zero})$
  - $(P \ \text{One})$

The type $\text{term}$ is the smallest type containing $\text{Zero}$ and $\text{One}$, and closed under $\text{Plus}$ and $\text{Mult}$. 
Recursive types: structural induction

To prove that for \( P : \text{term} \rightarrow \text{Prop} \), the theorem \( \forall t : \text{term}, P \ t \) holds, it is sufficient to:

- Prove that the property holds for the base cases:
  - \( P \) \ Zero
  - \( P \) \ One

- Prove that the property is transmitted inductively:
  - \( \forall t_1 t_2 : \text{term}, P \ t_1 \rightarrow P \ t_2 \rightarrow P \ (\text{Plus} \ t_1 t_2) \)
  - \( \forall t_1 t_2 : \text{term}, P \ t_1 \rightarrow P \ t_2 \rightarrow P \ (\text{Mult} \ t_1 t_2) \)

The type \( \text{term} \) is the smallest type containing \( \text{Zero} \) and \( \text{One} \), and closed under \( \text{Plus} \) and \( \text{Mult} \).
To prove that for $P : \text{term} \rightarrow \text{Prop}$, the theorem $\forall t : \text{term}, P t$ holds, it is sufficient to:

- Prove that the property holds for the base cases:
  - $(P \, \text{Zero})$
  - $(P \, \text{One})$

- Prove that the property is transmitted inductively:
  - $\forall t_1 \, t_2 : \text{term}, P \, t_1 \rightarrow P \, t_2 \rightarrow P \, (\text{Plus} \, t_1 \, t_2)$
  - $\forall t_1 \, t_2 : \text{term}, P \, t_1 \rightarrow P \, t_2 \rightarrow P \, (\text{Mult} \, t_1 \, t_2)$

The type $\text{term}$ is the smallest type containing $\text{Zero}$ and $\text{One}$, and closed under $\text{Plus}$ and $\text{Mult}$. 
Recursive types: structural induction

The induction principles generated at definition time by the system allow to:

- Program by recursion (Fixpoint)
- Prove by induction (induction)
We can compute some information on the size of a term:

\[
\text{Fixpoint height} \ (t : \text{natBinTree}) : \text{nat} := \\
\begin{align*}
&\text{match } t \text{ with} \\
&\quad | \text{Leaf } _{\_} => 0 \\
&\quad | \text{Node } _{\_} \ t_{1} \ t_{2} => \text{Max.max} (\text{height} \ t_{1}) (\text{height} \ t_{2}) + 1
\end{align*}
\]
Recursive types: program by structural induction

We can compute some information on the size of a term:

Fixpoint height (t : natBinTree) : nat :=
  match t with
  | Leaf _ => 0
  | Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.

Fixpoint size (t : natBinTree) : nat :=
  match t with

We can compute some information on the size of a term:

**Fixpoint** height (t : natBinTree) : nat :=
  match t with
  | Leaf _ => 0
  | Node _ t1 t2 => Max.max (height t1) (height t2) + 1
  end.

**Fixpoint** size (t : natBinTree) : nat :=
  match t with
  | Leaf _ => 1
  end.
Recursive types: program by structural induction

We can compute some information on the size of a term:

\[
\text{Fixpoint height (t : natBinTree) : nat :=}
\begin{align*}
\text{match t with} \\
| \text{Leaf } _ & \Rightarrow 0 \\
| \text{Node } _ t1 \ t2 & \Rightarrow \text{Max.max (height } t1) \ (\text{height } t2) + 1
\end{align*}
\]

\[
\text{Fixpoint size (t : natBinTree) : nat :=}
\begin{align*}
\text{match t with} \\
| \text{Leaf } _ & \Rightarrow 1 \\
| \text{Node } _ t1 \ t2 & \Rightarrow \text{(size } t1) + \text{(size } t2) + 1
\end{align*}
\]
Recursive types: program by structural induction

We can access some information contained in a term:

Require Import List.
Fixpoint label_at_occ (dflt : nat) (t : natBinTree)(u : list bool) :=
match u, t with
| nil, _ =>
    (match t with Leaf n => n | Node n _ _ => n end)
| b :: tl, t =>
    match t with
    | Leaf _ => dflt
    | Node _ t1 t2 =>
      if b then label_at_occ dflt t2 tl
      else label_at_occ dflt t1 tl
    end
end.
Recursive types: proofs by structural induction

We have already seen induction at work on nats and lists. Here it's goes on binary trees:

Lemma le_height_size : forall t : natBinTree, height t <= size t.

Proof.
induction t; simpl.
  auto.
apply plus_le_compat_r.
apply max_case.
  apply (le_trans _ _ _ IHt1).
  apply le_plus_l.
  apply (le_trans _ _ _ IHt2).
  apply le_plus_r.
Qed.
Structure of the definition of a recursive function

Inductive btree : Type := Leaf : btree
    | Node : btree -> btree -> btree.

Fixpoint get_subtree
    (l:list bool) (t:btree) {struct t} : btree :=
    match t, l with
    | Empty, _ => Empty
    | Node _ _, nil => t
    | Node tl tr, b :: l' =>
        if b then get_subtree l' tl else get_subtree l' tr
    end.

▶ Note the recursive calls made on \textit{tl} and \textit{tr}.
▶ The recursive call should be done on a strict sub-term of the argument.
▶ This ensures the termination of recursive functions
The termination of recursive functions is one of the component which ensures the logical consistency of Coq.
The termination of recursive functions is one of the components which ensures the logical consistency of Coq.

We have to live with this . . .
The termination of recursive functions is one of the components which ensures the logical consistency of Coq.

We have to live with this . . .

And we have to convince the system that all the functions we write are terminating.
An example of recursive function: \texttt{fact}

Recursive call should be made on strict sub-term:

\begin{verbatim}
Fixpoint fact n :=
    match n with
    | O => 1
    | S n' => n * fact n'
end.
\end{verbatim}

\begin{verbatim}
Definition fact' :=
    fix fact1 n :=
        match n with
        | O => 1
        | S n' => n * fact1 n'
    end.
\end{verbatim}
An example of recursive function: div2

Recursive call can be done on not immediate sub-terms:

```
Fixpoint div2 n :=
  match n with
  | S (S n') => S (div2 n')
  | _ => 0
  end.
```

A sub-term of strict sub-term is a strict sub-term
More general recursive calls

- It is possible to have recursive calls on results of functions.
- All cases must return a strict sub-term.
- Strict sub-terms may be obtained by applying functions on strict sub-terms.
  - This functions should only return sub-terms of their arguments. (not necessarily strict ones).
  - The system checks by looking at all cases.
Example of function that returns a sub-term

Definition pred (n : nat) :=
  match n with
  | 0 => n
  | S p => p
  end.

▶ In the 0 branch, the value is $n$, a (non-strict) sub-term of $n$.
▶ In the $S \ p$ branch, the value is $n$ a (strict) sub-term of $n$. 
Recursive function using \texttt{pred}

\begin{verbatim}
Fixpoint div2' (n : nat) :=
    match n with
    | 0 => n
    | S p => S (div2' (pred p))
end.
\end{verbatim}

The same trick can be played with \texttt{minus} which returns a sub-term of its first argument, to define euclidian division.
Mutual recursion

It is possible to define function by mutual recursion:

``` Occam
Fixpoint even n :=
  match n with
  | 0 => true
  | S n' => odd n'
  end
with odd n :=
  match n with
  | 0 => false
  | S n' => even n'
  end.
```
Lexicographic order

Sometimes termination functions is ensured by a lexicographic order on arguments. In Ocaml we can program:

```ocaml
let rec merge l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1::l1', x2::l2' ->
    if x1 <= x2 then
      x1 :: merge l1' l2
    else
      x2 :: merge l1 l2';;
```

There are two recursive calls `merge l1' l2` and `merge l1 l2'`. 
Lexicographic order

Sometimes termination functions is ensured by a lexicographic order on arguments. In Ocaml we can program:

```ocaml
let rec merge l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1::l1', x2::l2' ->
    if x1 <= x2 then
      x1 :: merge l1' l2
    else
      x2 :: merge l1 l2';;
```

There are two recursive calls `merge l1' l2` and `merge l1 l2'`. 
Solution in Coq: internal recursion

Coq also makes it possible to describe *anonymous* recursive function. Sometimes necessary to use them for difficult recursion patterns.

```
Fixpoint merge (l1 l2:list nat) : list nat :=
  match l1, l2 with
  | nil, _ => l2 | _, nil => l1
  | x1::l1', x2::l2' =>
    if leb x1 x2 then x1::merge l1' l2
    else
      x2 :: (fix merge_aux (l2:list nat) :=
        match l2 with
        | nil => l1
        | x2::l2' =>
          if leb x1 x2 then x1::merge l1' l2
          else x2:: merge_aux l2'
        end) l2'
  end.
```
The style is not very readable (use the Section instead)
Fixpoint merge l1 l2 :=
  let fix merge_aux l2 :=
    match l1, l2 with
    | nil, _ => l2
    | _, nil => l1
    | x1::l1’, x2::l2’ =>
      if leb x1 x2 then x1::merge l1’ l2
        else x2::merge_aux l2’
    end
  in merge_aux l2.

Compute merge (2::3::5::7::nil) (3::4::10::nil).
= 2 :: 3 :: 3 :: 4 :: 5 :: 7 :: 10 :: nil
  : list nat
More general recursion

- Constraints of structural recursion may be too cumbersome.
- Sometimes a measure decreases, which cannot be expressed by structural recursion.
- The general solution provided by well-founded recursion.
- An intermediate solution provided by the command Function.
Example using Function: fact on Z

Integers have a more complex structure than natural numbers

Inductive positive : Set :=
    | xH : positive (* encoding of 1 *)
    | xO : positive -> positive (* encoding of 2*p *)
    | xI : positive -> positive. (* encoding of 2*p+1 *)

Inductive Z : Set :=
    | Z0: Z | Zpos: positive -> Z | Zneg: positive -> Z.

- This type makes computation more efficient.
- \(x - 1\) is not a structural sub-term of \(x\).
- For instance 3 is \(Zpos \ (xI \ xH)\) and 2 is \(Zpos \ (xO \ xH)\).
Example using Function: fact on Z

Require Import Recdef.

Function factZ (x : Z) {measure Zabs_nat x} :=
  if Zle_bool x 0 then 1 else x * fact (x - 1).

1 subgoal

forall x : Z, Zle_bool x 0 = false ->
(Zabs_nat (x - 1) < Zabs_nat x)%nat

Now, we prove explicitly that the measure decreases.
Definition slen (p:list nat * list nat) :=
   length (fst p) + length (snd p).

Function Merge (p:list nat * list nat) { measure slen p } : list nat :=
match p with
  | (nil, l2) => l2
  | (l1, nil) => l1
  | ((x1::l1') as l1, (x2::l2') as l2) =>
      if leb x1 x2 then x1::Merge (l1',l2)
      else x2::Merge (l1,l2')
end.

(* Two goals *)
...
Defined.

Compute Merge (2::3::5::7::nil, 3::4::10::nil).
Well-founded Relations

Dotted lines represent any number of elementary relationships
Minimal elements are *accessible*
Elements whose all predecessors are accessible become accessible
Some time later …
How to encode well founded relations in Coq? By crafting the type of trees with no infinite branch.

Let’s try.
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive btree : Type :=}
\]
\[
| \text{Leaf : btree} \\
| \text{Node : btree -> btree -> btree.}
\]

A type for finitly branching trees:

\[
\text{Inductive ntree : Type :=}
\]
\[
| \text{Leaf : ntree} \\
| \text{Node : (list ntree) -> ntree.}
\]

A type for countably branching trees:

\[
\text{Inductive itree : Type :=}
\]
\[
| \text{Leaf : itree} \\
| \text{Node : (nat -> itree) -> itree.}
\]
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive } \text{btree} : \text{Type} :=
\]

A type for finitly branching trees:

\[
\text{Inductive } \text{ntree} : \text{Type} :=
\]

A type for countably branching trees:

\[
\text{Inductive } \text{itree} : \text{Type} :=
\]
Well founded relations in Coq

A type for binary trees:

\[
\begin{align*}
\text{Inductive } \text{btree} : \text{Type} :&= \\
&| \text{Leaf} : \text{btree}
\end{align*}
\]
Well founded relations in Coq

A type for binary trees:

\textbf{Inductive btree : Type :=}
  \textbf{| Leaf : btree}
  \textbf{| Node : btree -> btree -> btree.}
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive btree : Type :=}
\begin{align*}
& \text{ | Leaf : btree} \\
& \text{ | Node : btree -> btree -> btree.}
\end{align*}
\]

A type for finitly branching trees:

\[
\text{Inductive ntree : Type :=}
\begin{align*}
& \text{ | Leaf : ntree} \\
& \text{ | Node : (list ntree) -> ntree.}
\end{align*}
\]

A type for countably branching trees:

\[
\text{Inductive itree : Type :=}
\begin{align*}
& \text{ | Leaf : itree} \\
& \text{ | Node : (nat -> itree) -> itree.}
\end{align*}
\]
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive } \text{btree} : \text{Type} := \\
| \text{Leaf} : \text{btree} \\
| \text{Node} : \text{btree} \to \text{btree} \to \text{btree}.
\]

A type for finitly branching trees:

\[
\text{Inductive } \text{ntree} : \text{Type} := \\
| \text{Leaf} : \text{ntree}
\]
Well founded relations in Coq

A type for binary trees:

```
Inductive btree : Type :=
  | Leaf : btree
  | Node : btree -> btree -> btree.
```

A type for finitly branching trees:

```
Inductive ntree : Type :=
  | Leaf : ntree
  | Node : (list ntree) -> ntree.
```

A type for countably branching trees:

```
Inductive itree : Type :=
  | Leaf : itree
  | Node : (nat -> itree) -> itree.
```
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive } \text{btree} : \text{Type} := \\
\quad | \text{Leaf} : \text{btree} \\
\quad | \text{Node} : \text{btree} \to \text{btree} \to \text{btree}.
\]

A type for finitly branching trees:

\[
\text{Inductive } \text{ntree} : \text{Type} := \\
\quad | \text{Leaf} : \text{ntree} \\
\quad | \text{Node} : (\text{list ntree}) \to \text{ntree}.
\]
Well founded relations in Coq

A type for binary trees:

\[
\text{Inductive btree : Type :=}
\begin{align*}
| \text{Leaf} : & \text{btree} \\
| \text{Node} : & \text{btree} \to \text{btree} \to \text{btree}.
\end{align*}
\]

A type for finitly branching trees:

\[
\text{Inductive ntree : Type :=}
\begin{align*}
| \text{Leaf} : & \text{ntree} \\
| \text{Node} : & \text{(list ntree)} \to \text{ntree}.
\end{align*}
\]

A type for countably branching trees:
A type for binary trees:

Inductive btree : Type :=
  | Leaf : btree
  | Node : btree -> btree -> btree.

A type for finitly branching trees:

Inductive ntree : Type :=
  | Leaf : ntree
  | Node : (list ntree) -> ntree.

A type for countably branching trees:

Inductive itree : Type :=
  | Leaf : itree
  | Node
Well founded relations in Coq

A type for binary trees:

Inductive btree : Type :=
  | Leaf : btree
  | Node : btree -> btree -> btree.

A type for finitly branching trees:

Inductive ntree : Type :=
  | Leaf : ntree
  | Node : (list ntree) -> ntree.

A type for countably branching trees:

Inductive itree : Type :=
  | Leaf : itree
  | Node : (nat -> itree) -> itree.
Well founded relations in Coq

A type for countably branching trees:

\[
\text{Inductive itree : Type := } \\
| \text{Leaf : itree} \\
| \text{Node : (nat -> itree) -> itree.}
\]
A type for countably branching trees:

\[
\text{Inductive itree : Type :=}
\begin{align*}
& \mid \text{Leaf : itree} \\
& \mid \text{Node : (nat \to itree) \to itree}.
\end{align*}
\]

We can still program with inhabitants of that type:

\[
\text{Fixpoint ileft t :=}
\begin{align*}
& \text{match t with} \\
& \mid \text{ILeaf} \Rightarrow t \\
& \mid \text{INode f} \Rightarrow \text{ileft (f 0)}
\end{align*}
\]
end.
A (dependent) type for trees with bounded degree:

\[
\text{Inductive \ dtree'} : \text{nat} \to \text{Type} :=
\]

Well founded relations in Coq
A (dependent) type for trees with bounded degree:

Inductive dtree' : nat -> Type :=
  | Leaf' : forall n, dtree' n

In fact the \texttt{Leaf} constructor can be removed:

Inductive dtree : nat -> Type :=
  | Node' : forall n, (forall m, m < n -> dtree m) -> dtree n.

Because we can construct a \texttt{(DLeaf : dtree 0)} (exercise).
Well founded relations in Coq

A (dependent) type for trees with bounded degree:

Inductive dtree’ : nat -> Type :=
  | Leaf’ : forall n, dtree’ n
  | Node’
A (dependent) type for trees with bounded degree:

Inductive dtree' : nat -> Type :=
| Leaf' : forall n, dtree' n
| Node' : forall n, (forall m, m < n -> dtree' m) -> dtree' n.
A (dependent) type for trees with bounded degree:

\[
\text{Inductive } \text{dtree'} : \text{n} \rightarrow \text{Type } := \\
\quad | \text{Leaf'} : \forall \text{n}, \text{dtree'} \text{n} \\
\quad | \text{Node'} : \forall \text{n}, \\
\quad \quad (\forall \text{m}, \text{m} < \text{n} \rightarrow \text{dtree'} \text{m}) \rightarrow \text{dtree'} \text{n}.
\]

In fact the Leaf constructor can be removed:
A (dependent) type for trees with bounded degree:

Inductive dtree’ : nat -> Type :=
    | Leaf’ : forall n, dtree’ n
    | Node’ : forall n,
      (forall m, m < n -> dtree’ m) -> dtree’ n.

In fact the Leaf constructor can be removed:

Inductive dtree : nat -> Type :=
A (dependent) type for trees with bounded degree:

\[
\text{Inductive } \text{dtree}' : \text{nat} \rightarrow \text{Type} := \\
| \text{Leaf}' : \forall n, \text{dtree}' n \\
| \text{Node}' : \forall n, \\
(\forall m, m < n \rightarrow \text{dtree}' m) \rightarrow \text{dtree}' n.
\]

In fact the Leaf constructor can be removed:

\[
\text{Inductive } \text{dtree} : \text{nat} \rightarrow \text{Type} := \\
| \text{Node}
\]
A (dependent) type for trees with bounded degree:

\[
\text{Inductive \ dtree' : nat \rightarrow Type :=}
\]
\[
| \text{Leaf'} : \forall n, \, \text{dtree'} n \\
| \text{Node'} : \forall n, \\
\quad (\forall m, \, m < n \rightarrow \text{dtree'} m) \rightarrow \text{dtree'} n.
\]

In fact the Leaf constructor can be removed:

\[
\text{Inductive \ dtree : nat \rightarrow Type :=}
\]
\[
| \text{Node} : \forall n, \\
\quad (\forall m, \, m < n \rightarrow \text{dtree m}) \rightarrow \text{dtree n}.
\]
A (dependent) type for trees with bounded degree:

Inductive dtree' : nat -> Type :=
    | Leaf' : forall n, dtree' n
    | Node' : forall n,
            (forall m, m < n -> dtree' m) -> dtree' n.

In fact the Leaf constructor can be removed:

Inductive dtree : nat -> Type :=
    | Node  : forall n,
            (forall m, m < n -> dtree m) -> dtree n.

Because we can construct a (DLeaf : dtree 0) (exercise).
A (dependent) type for trees with bounded degree:

\[
\text{Inductive } \text{dtree} : \text{nat} \rightarrow \text{Type} := \\
| \text{Node} : \forall n, \\
(\forall m, m < n \rightarrow \text{dtree} m) \rightarrow \text{dtree} n.
\]
Well founded relations in Coq

A (dependent) type for trees with bounded degree:

Inductive dtree : nat -> Type :=
  | Node : forall n,
     (forall m, m < n -> dtree m) -> dtree n.

We can generalize to a binary relation $R$ on $\mathbb{N}$:

Inductive atree (R : nat -> nat -> Prop) : nat -> Type :=
  | ANode : forall n,
     (forall m, R m n -> atree R m) -> atree R n.
Well founded relations in Coq

A (dependent) type for trees with bounded degree:

\[
\text{Inductive } \text{dtree} : \text{nat} \rightarrow \text{Type} := \\
| \text{Node} : \forall n, \\
(\forall m, m < n \rightarrow \text{dtree} m) \rightarrow \text{dtree} n.
\]

We can generalize to a binary relation \( R \) on \( \text{nat} \):

\[
\text{Inductive } \text{atree} (R : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}) : \text{nat} \rightarrow \text{Type} := \\
| \text{ANode} : \forall n, \\
(\forall m, R m n \rightarrow \text{atree} R m) \rightarrow \text{atree} R n.
\]

If it satisfies the following, this relation is for sure well founded:

\[
\text{Definition } \text{nat\_well\_founded} (R : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop}) := \\
\forall n, \text{atree} R n.
\]
A relation is well founded if all elements are accessible.

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

Definition well_founded (A:Type) (R:A->A->Prop) :=
  forall a, Acc R a.

It is possible to define functions by recursion on the accessibility proof of an element (Function, Program are based on this).
Coq’s Standard Library provides us with some useful examples of well-founded relations:

- The predicate `lt` over `nat` (but you can use `measure` instead)
- The predicate `Zwf c`, which is the restriction of `<` to the interval `[c, ∞[ of \( \mathbb{Z} \).

Libraries Relations, Wellfounded contains (dependent) cartesian product, transitive closure, lexicographic product and exponentiation.
More examples: log10

Function log10 \( (n : \mathbb{Z}) \{\text{wf} (\text{Zwf} 1) n\} : \mathbb{Z} := \)
\[
\text{if } \text{Zlt\_bool } n 10 \text{ then } 0 \text{ else } 1 + \text{log10} (n / 10).
\]

Proof.

(* first goal *)
intros n Hleb.
unfold Zwf.
generalize (Zlt_cases n 10) (Z\_div\_lt n 10);rewrite Hleb.
omega.

(* Second goal *)
apply Zwf\_well\_founded.
Defined.

(* Compute log10 2. : and wait (for a long time) ... *)
Function log10 (n : Z) {measure Zabs_nat n} : Z :=
    if Zlt_bool n 10 then 0 else 1 + log10 (n / 10).
Proof.
    (* first goal *)
    intros n Hleb.
    unfold Zwf; generalize (Zlt_cases n 10); rewrite Hleb; intros Hle.
    apply Zabs_nat_lt.
    split.
    apply Z_div_pos; omega.
    apply Zdiv_lt_upper_bound; omega.
Defined.
Generating one’s own induction principle

Sometime, the generated induction principle is not usable.

Inductive tree (A:Type) :=
  \mid \text{Node} : A -> \text{list \ (tree A)} -> \text{tree A}.

Check tree_ind.

\text{tree_ind}
  : \forall (A : \text{Type}) (P : \text{tree A} -> \text{Prop}),
    (\forall (a : A) (l : \text{list \ (tree A)}), P (\text{Node} A a l))
  \forall t : \text{tree A}, P t
Generating one’s own induction principle

my_tree_ind : forall (A : Type)
  (P : tree A -> Prop) (Pl : list (tree A) -> Prop),
  (forall a l, Pl l -> P (Node _ a l)) ->
  Pl nil ->
  (forall t l, P t -> Pl l -> Pl (t :: l)) ->
  forall t, P t

This is a good exercise...
Principles of coinductive definitions

- 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: $\forall X, (X \subseteq FX) \rightarrow X \subseteq \nu X. F X$
  - 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.
Example: streams

Variable A : Type.

CoInductive Stream : Type :=
  Cons : A -> Stream -> Stream.

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

Lemma cte_hd : forall a, hd (cte a) = a.
Proof. trivial. Qed.

Lemma cte_tl : forall a, tl (cte a) = cte a.
Proof. trivial. Qed.

Lemma cte_eq : forall a, cte a = Cons a (cte a).
  Proof.
  intros.
  transitivity (Cons (hd (cte a)) (tl (cte a)));
  trivial.
  now case (cte a); auto.
Qed.
Functions should also be guarded

Filter on stream

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

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

Might introduce a closed term of type Stream which does not reduce to a constructor.
Coinductive family

Notion of infinite proof:

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

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

Definition of an extentional (bisimulation) equality predicate:

CoInductive eqS (s t:Stream) : Prop :=
  eqS_intros : hd s = hd t \rightarrow eqS (tl s) (tl t) \rightarrow 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).
A CS example

The computation monad (Megacz – PLPV’07, ...):

CoInductive comp (A : Type) :=
| Done (a : A) : comp A
| Step (c : comp A) : comp A

One Step is one “tick” of a computation.
Exercise: Show it is a monad, with special action:

    eval : forall A, comp A -> nat -> option A

What’s the right notion of equality on computations?
Write the Collatz function using this monad: