Recap

\[ A \rightarrow B \equiv \forall x : A, \ B \]

when

\[ x \notin \text{FVar}(B) \]
Recap

- definition of functions
- `fun x => M` notation for anonymous functions
- functional kernel of Coq is a typed \(\lambda\)-calculus
- all calculations are finite
- every Coq term has a unique normal form
- Enumerated (finite) types
Plan

• recap
• recursive types
• recursive definitions
• example 1: natural numbers
• example 2: day lists
• example 3: binary trees
Recap’

- **Definition** for functions definitions
- **Check** to show types
- **Compute** to show values
- **Eval compute in** to show values
- **Inductive** to define a new data type
- **match ... with** for case analysis on constructors
- **Type** set of all types
- **simpl** to compute normal form
- **reflexivity** to conclude with trivial equality
- **discriminate** to conclude with distinct constructors

**Example** neq_on_days : monday <> tuesday.
Proof. **discriminate**. Qed.
typed λ-calculus with recursion (and a bit of arithmetic)
PCF language (1/3)

- Terms

  \( M, N, P \quad ::= \quad x, y, z, \ldots \quad \text{(variables)} \)
  
  \mid \quad \lambda x. M \quad \text{(M as function of x)}
  
  \mid \quad M \ (N) \quad \text{(M applied to N)}
  
  \mid \quad n \quad \text{(natural integer constant)}
  
  \mid \quad M \otimes N \quad \text{(arithmetic op)}
  
  \mid \quad \text{if} z \ N \text{then} M \text{else} N \quad \text{(conditionnal)}
  
  \mid \quad Y \quad \text{(recursion)}
PCF language (1/3)

- Terms

\[ M, N, P \quad ::= \quad x, y, z, \ldots \quad \text{(variables)} \]
\[ \quad \mid \lambda x.M \quad \text{(M as function of x)} \]
\[ \quad \mid M(N) \quad \text{(M applied to N)} \]
\[ \quad \mid n \quad \text{(natural integer constant)} \]
\[ \quad \mid M \otimes N \quad \text{(arithmetic op)} \]
\[ \quad \mid \text{if } z \ P \text{ then } M \text{ else } N \quad \text{(conditionnal)} \]
\[ \quad \mid Y \quad \text{(recursion)} \]

- Calculations (“reductions”)

\((\lambda x. M)(N) \rightarrow M \{ x := N \} \)
\[ m \otimes n \rightarrow m \otimes n \]
\[ \text{ifz } 0 \text{ then } M \text{ else } N \rightarrow M \]
\[ \text{ifz } n + 1 \text{ then } M \text{ else } N \rightarrow N \]
\[ Yf \rightarrow f(Yf) \]

[Plotkin 1975]
Thus following term:

\[(\lambda \text{Fact}. \text{Fact}(3))\]

\[(Y(\lambda f. \lambda x. \text{if} z \times \text{then} 1 \text{ else } x \times f(x - 1)))\]
3 * (2*(1*(if z 0 then 1 else ((3-1)-1) * (λx.λy.x if z then 1 else x * f(x-1))(xx))(λx.λy.x if z then 1 else x * f(x-1))(xx)((3-1)-1)-1)))
PCF language (3/3)

• Some computations terminate, but not all. (normalization, but not strong normalization)

Let $F = \lambda f. \lambda x. \text{if } z \text{ then } 1 \text{ else } x \ast f(x - 1)$. Then

$$(\lambda \text{Fact}. \text{Fact}(3))(YF) \rightarrow \rightarrow \ldots \rightarrow 6$$

$$\rightarrow (\lambda \text{Fact}. \text{Fact}(3))(F(YF))$$

$$\rightarrow (\lambda \text{Fact}. \text{Fact}(3))(F(F(YF)))$$

$$\rightarrow \ldots$$

$$\rightarrow (\lambda \text{Fact}. \text{Fact}(3))(F^n(YF))$$

$$\rightarrow \ldots$$

• Quite common in usual programming languages

• In Coq, we do have strong normalization.
• In Coq, we do have strong normalization.
Computability

• Any most general model of computation has non terminating programs.
  [Kleene, 1950]

• Coq cannot express all computable functions

• but the power of Coq typing allows many of them
Recursive data types
Recursive types (1/7)

Inductive nat : Set :=
  | 0 : nat
  | S : nat -> nat.

Inductive daylist : Type :=
  | nil : daylist
  | cons : day -> daylist -> daylist.

Base case constructors do not feature self-reference to the type. Recursive case constructors do.

Definition weekend_days := cons saturday (cons sunday nil)).
Recursive types (2/7)

- \(0, 1 = S(0), 2 = S(S(0)), 3 = S(S(S(0)), \ldots\) (unary representation)

- \text{cons tuesday (cons wednesday (cons friday (cons sunday nil)))}
Recursive types (3/7)

... Coq language can handle notations for infix operators.

Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[]" := nil.
Notation "[ x , .. , y ]" := (cons x .. (cons y nil) ..).

Notation "x + y" := (plus x y)
(at level 50, left associativity).

Therefore \texttt{weekend\_days} can be also written:

Definition weekend\_days := saturday :: sunday :: nil.

or

Definition weekend\_days := [saturday, sunday].
Recursive types (4/7)

... with recursive definitions of functions.

Fixpoint length (l:daylist) {struct l} : nat :=
  match l with
  | nil => 0
  | d :: l' => S (length l')
  end.

Fixpoint repeat (d:day) (count:nat) {struct count} : daylist :=
  match count with
  | 0 => nil
  | S count' => d :: (repeat d count')
  end.

The decreasing argument is precisely as hint for termination.

to insure strong normalization
Recursive types (5/7)

... with recursive definitions of functions.

Fixpoint app (l1 l2 : daylist) {struct l1} : daylist :=
  match l1 with
  | nil => l2
  | d :: t => d :: (app t l2)
end.

Notation "x ++ y" := (app x y)
  (right associativity, at level 60).

Example test_app1: [monday,tuesday,wednesday] ++ [thursday,friday] = 
  [monday,tuesday,wednesday,thursday,friday].
Proof. reflexivity. Qed.

Example test_app2: nil ++ [monday,wednesday] = [monday,wednesday].
Proof. reflexivity. Qed.

Example test_app3: [monday,wednesday] ++ nil = [monday,wednesday].
Proof. reflexivity. Qed.
Recursive types (5/7)

... with recursive definitions of functions.

Fixpoint app (l1 l2 : daylist) \{struct l1\} : daylist :=
match l1 with
| nil => l2
| d :: t => d :: (app t l2)
end.

Notation "x ++ y" := (app x y)
(right associativity, at level 60).

Example test_app1: [monday,tuesday,wednesday] ++ [thursday,friday] =
[monday,tuesday,wednesday,thursday,friday].
Proof. reflexivity. Qed.

Example test_app2: nil ++ [monday,wednesday] = [monday,wednesday].
Proof. reflexivity. Qed.

Example test_app3: [monday,wednesday] ++ nil = [monday,wednesday].
Proof. reflexivity. Qed.
Recursive types (5/7)

... with recursive definitions of functions.

Fixpoint app (l1 l2 : daylist) {struct l1} : daylist :=
match l1 with
| nil => l2
| d :: t => d :: (app t l2)
end.

Notation "x ++ y" := (app x y)
(right associativity, at level 60).

Example test_app1: [monday,tuesday,wednesday] ++ [thursday,friday] =
[monday,tuesday,wednesday,thursday,friday].
Proof. reflexivity. Qed.

Example test_app2: nil ++ [monday,wednesday] = [monday,wednesday].
Proof. reflexivity. Qed.

Example test_app3: [monday,wednesday] ++ nil = [monday,wednesday].
Proof. reflexivity. Qed.
Recursive types (6/7)

... with recursive definitions of functions.

Definition bag := daylist.

Definition eq_day (d:day)(d’:day) : bool :=
    match d, d’ with
    | monday, monday | tuesday, tuesday | wednesday, wednesday => true
    | thursday, thursday | friday, friday => true
    | saturday, saturday => true
    | sunday, sunday => true
    | _, _ => false
    end.

Fixpoint count (d:day) (s:bag) {struct s} : nat :=
    match s with
    | nil => 0
    | h :: t => if eq_day d h then 1 + count d t else count d t
    end.
Exercice 4 Show following propositions:

Example test_count1: count sunday [monday, sunday, friday, sunday] = 2.
Example test_count2: count sunday [monday, tuesday, friday, friday] = 0.

Exercice 5 Define union of two bags of days.

Exercice 6 Define add of one day to a bag of days.

Exercice 7 Define remove_one day from a bag of days.

Exercice 8 Define remove_all occurences of a day from a bag of days.

Exercice 9 Define member to test if a day is member of a bag of days.

Exercice 10 Define subset to test if a bag of days is a subset of another bag of days.
Remark on constructors

- Constructors are **injective**:

  Lemma inj_succ : forall n m, S n = S m -> n = m.
  Proof.
  intros n m H.
  injection H.
  easy.
  Qed.

- Constructors are all **distinct**.
Recap

- **Definition** for functions definitions
- **Check** to show types
- **Compute** to show values
- **Eval compute in** to show values
- **Inductive** to define a new data type
- **match ... with** for case analysis on constructors
- **Type** set of all types
- **simpl** to compute normal form
- **reflexivity** to conclude with trivial equality
- **discriminate** to conclude with distinct constructors
- **Fixpoint** for recursive functions definitions
- **struct** to hint for termination
Other recursive datatypes (1/2)

Another recursive type: binary trees.

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

Abstract Syntax Trees for terms.

Inductive term : Set :=
| Zero : term
| One : term
| Plus : term -> term -> term
Counting \textbf{leaves} and \textbf{nodes} in binary trees.

Fixpoint \texttt{count\_leaves} (t : natBinTree) \{struct t\} : nat :=  
match t with  
| leaf n => 1  
| node n t1 t2 => (count\_leaves t1) + (count\_leaves t2)  
end.

Fixpoint \texttt{count\_nodes} (t : natBinTree) \{struct t\} : nat :=  
match t with  
| leaf n => 0  
| node n t1 t2 => 1 + (count\_nodes t1) + (count\_nodes t2)  
end.