5th Asian-Pacific Summer School on Formal Methods
August 5-10, 2013, Tsinghua University, Beijing, China

Inductive data types (I)

jean-jacques.levy@inria.fr
2013-8-6

http://sts.thss.tsinghua.edu.cn/Coqschool2013

Notes adapted from
Assia Mahboubi
(coq school 2010, Paris) and
Benjamin Pierce (software
foundations course, UPenn)

Recap

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

Recap

$$A \to B \equiv \forall x{:}A,\ B$$

when

$$x \notin \mathrm{FVar}(B)$$

Plan

今天 小菜一碟

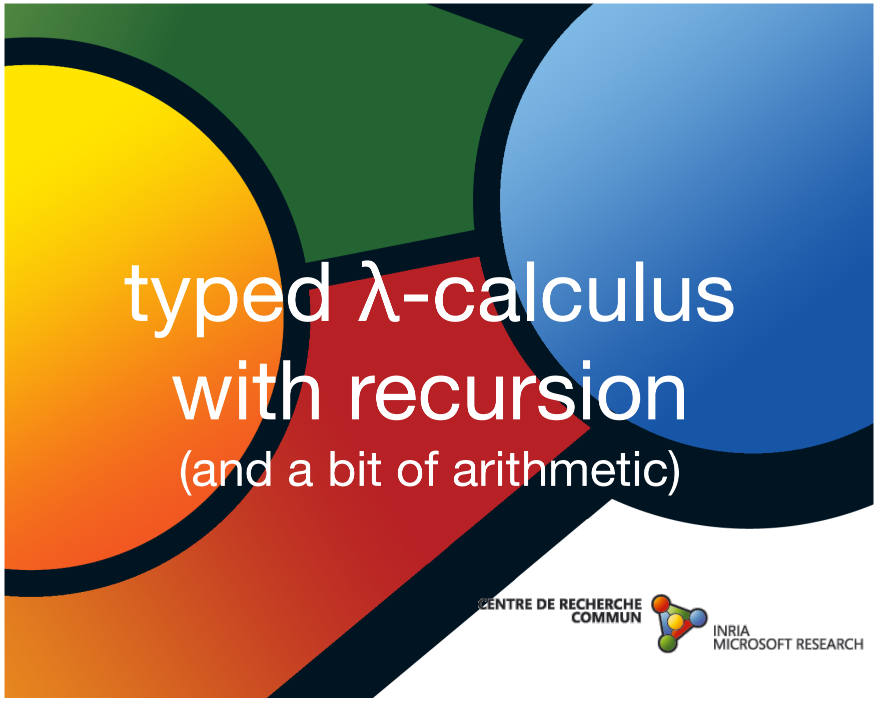
- recap
- recursive types
- recursive definitions
- example 1: natural numbers
- example 2: day lists
- example 3: binary trees

Recap'

- Coq commands / keywords:

– `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.
```



PCF language (1/3)

[Plotkin 1975]

- Terms

M, N, P	::=	$x, y, z, \ldots$	(variables)
	|	$\lambda x.M$	(M as function of x)
	|	$M\,(\,N\,)$	(M applied to N)
	|	$\underline{n}$	(natural integer constant)
	|	$M \otimes N$	(arithmetic op)
	|	$\texttt{ifz}\ P\ \texttt{then}\ M\ \texttt{else}\ N$	(conditionnal)
	|	Y	(recursion)

PCF language (1/3)

[Plotkin 1975]

- Terms

M, N, P	::=	$x, y, z, \ldots$	(variables)
	|	$\lambda x.M$	(M as function of x)
	|	$M\,(\,N\,)$	(M applied to N)
	|	$\underline{n}$	(natural integer constant)
	|	$M \otimes N$	(arithmetic op)
	|	$\texttt{ifz}\ P\ \texttt{then}\ M\ \texttt{else}\ N$	(conditionnal)
	|	Y	(recursion)

- Calculations ("reductions")

$$(\lambda x.M)(N) \longrightarrow M\{x := N\}$$

$$\underline{m} \otimes \underline{n} \longrightarrow \underline{m \otimes n}$$

$$\texttt{ifz}\ \underline{0}\ \texttt{then}\ M\ \texttt{else}\ N \longrightarrow M$$

$$\texttt{ifz}\ \underline{n+1}\ \texttt{then}\ M\ \texttt{else}\ N \longrightarrow N$$

$$Yf \longrightarrow f(Yf)$$

PCF language (2/3)

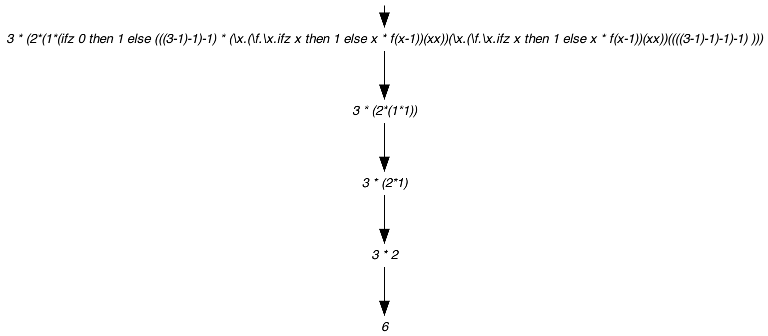
$\texttt{Fact}(3)$

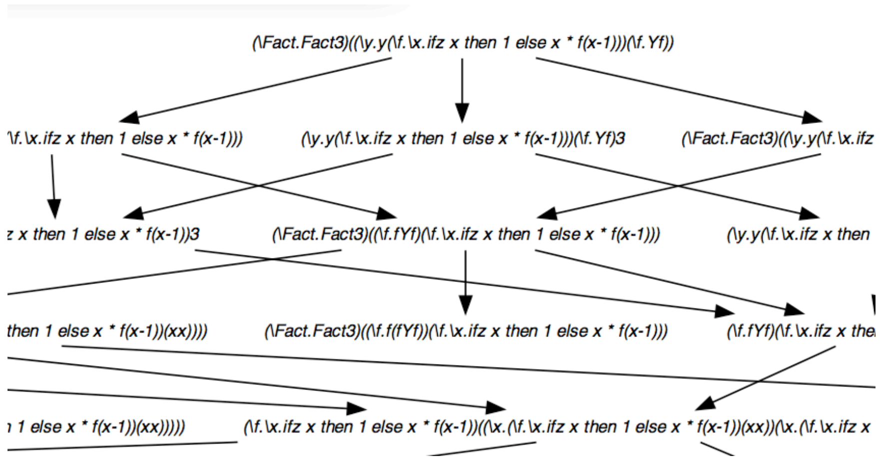
$\texttt{Fact} = Y(\lambda f.\lambda x.\ \texttt{ifz}\ x\ \texttt{then}\ 1\ \texttt{else}\ x \star f(x-1))$

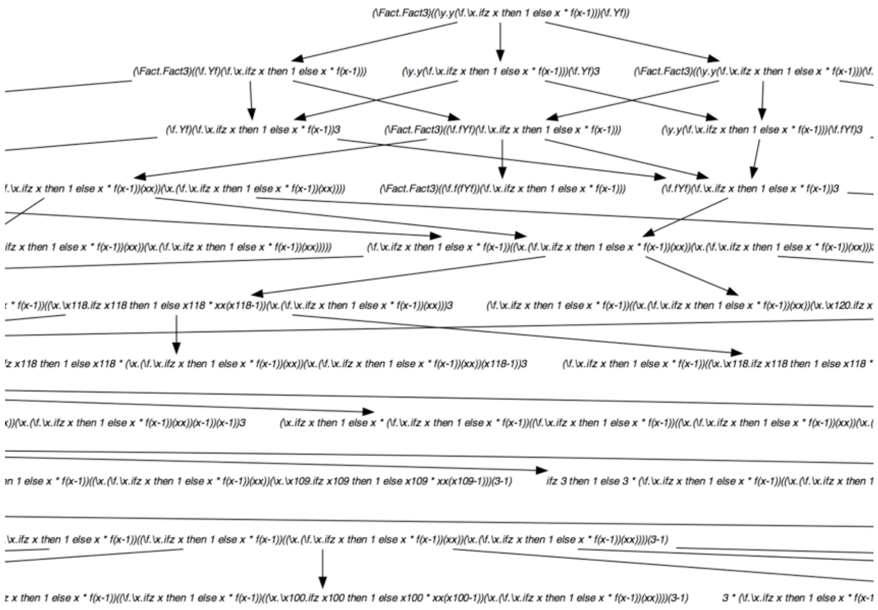
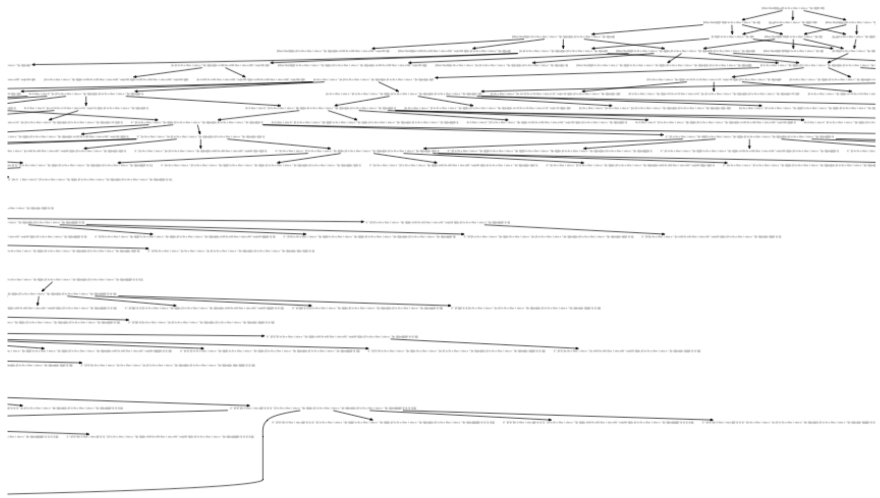
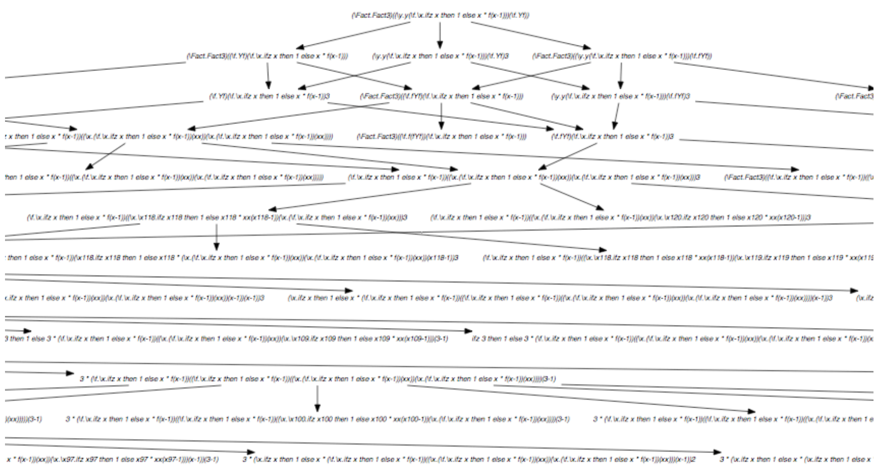
Thus following term:

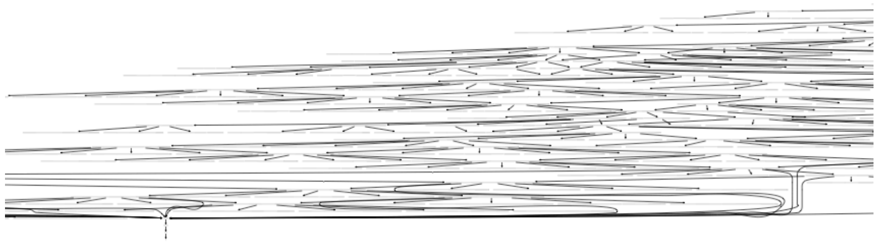
$(\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))$

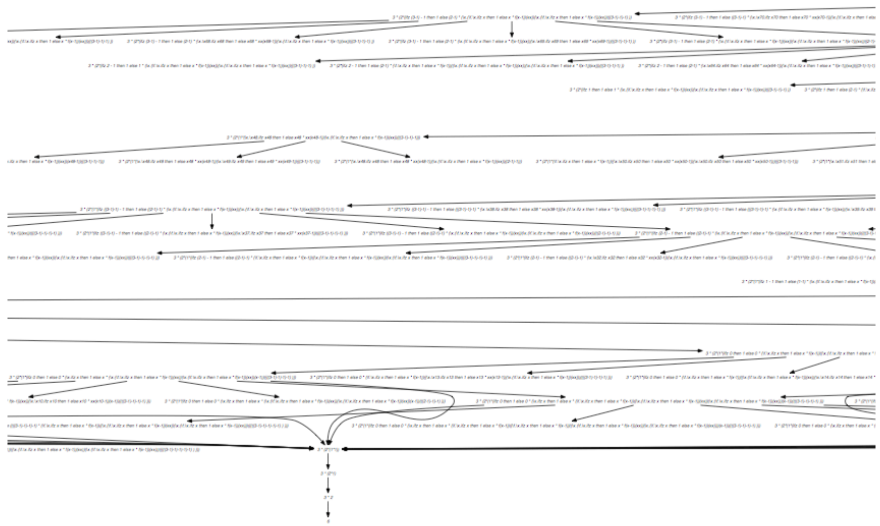
$\quad(Y(\lambda f.\lambda x.\ \texttt{ifz}\ x\ \texttt{then}\ 1\ \texttt{else}\ x \star f(x-1)))$

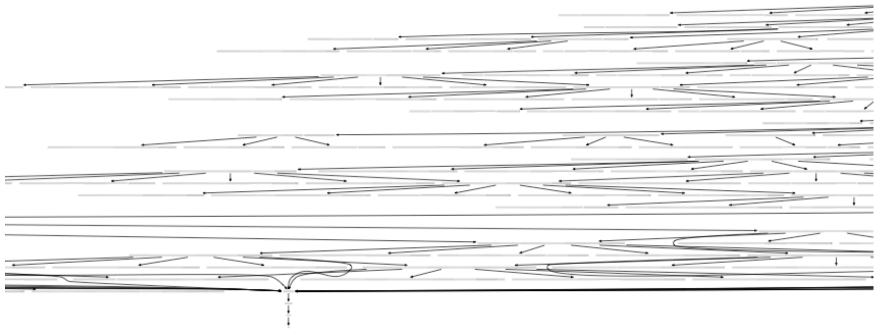


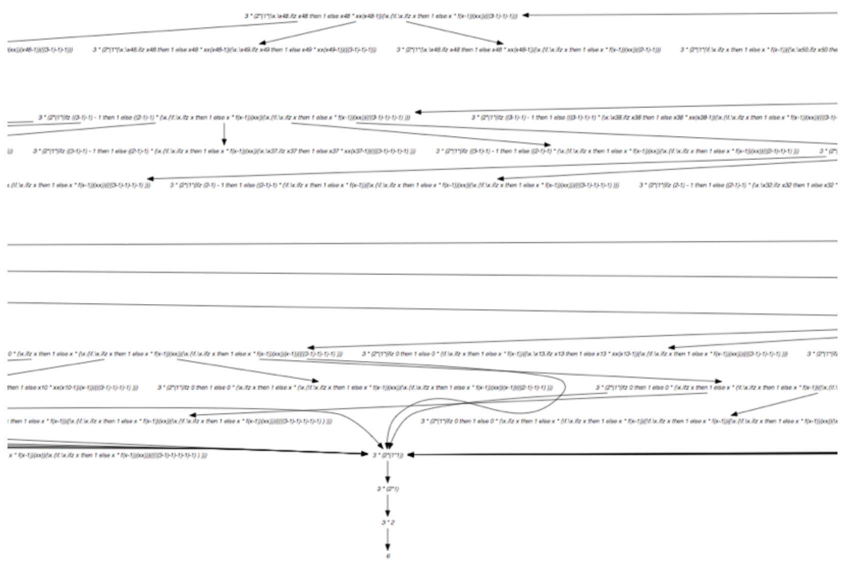



(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf))
(\Fact.Fact3)((\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf)3
(\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1))3
(\Fact.Fact3)((\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.fYf)3
(\Fact.Fact3)((\f.f(fYf))(\f.\x.ifz x then 1 else x * f(x-1)))
(\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1))3



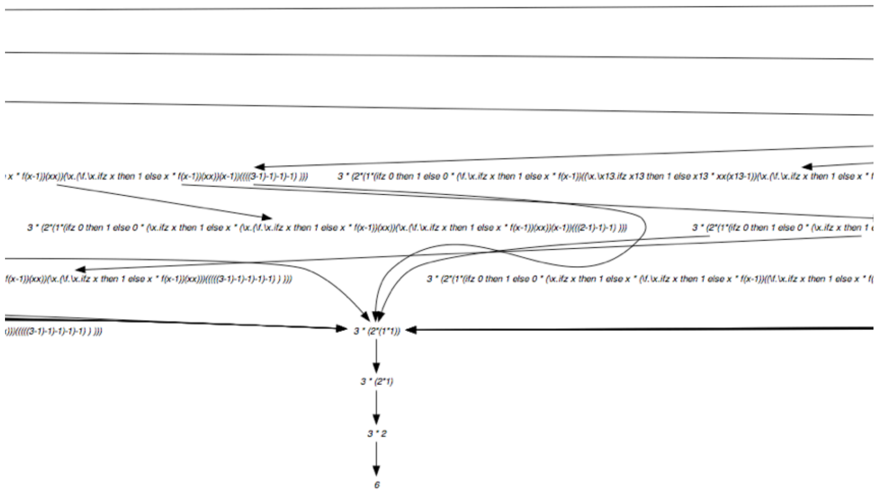









3 * (2*(1*1))
3 * (2*1)
3 * 2
6



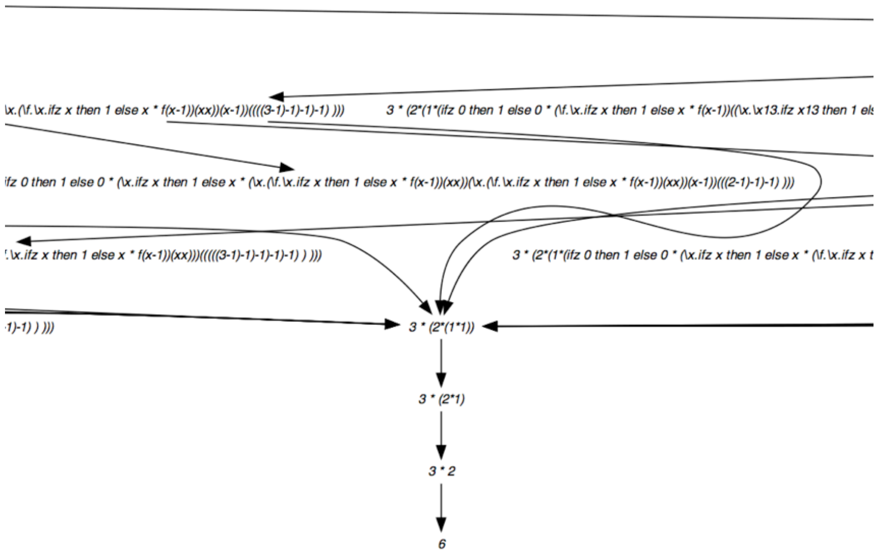
PCF language (3/3)

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

Let $F = \lambda f.\lambda x.\ \texttt{ifz}\ x\ \texttt{then}\ 1\ \texttt{else}\ x \star f(x-1)$. Then

$$(\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))\ (YF) \longrightarrow \longrightarrow \cdots\cdots \longrightarrow 6$$
$$\longrightarrow (\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))\ (F(YF))$$
$$\longrightarrow (\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))\ (F(F(YF)))$$
$$\longrightarrow \cdots$$
$$\longrightarrow (\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))\ (F^n(YF))$$
$$\longrightarrow \cdots$$

- Quite common in usual programming languages
- In Coq, we do have strong normalization.



PCF language (3/3)

- 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 types (1/7)

```
Inductive nat : Set :=
  | O : 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)).
```

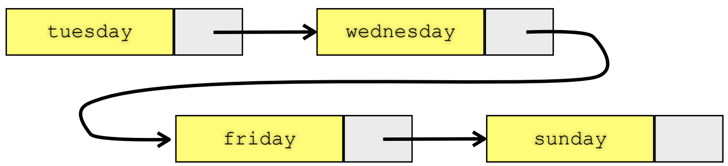
Recursive data types

CENTRE DE RECHERCHE COMMUN
INRIA
MICROSOFT RESEARCH

Recursive types (2/7)

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

- cons tuesday (cons wednesday
  (cons friday (cons sunday nil)))



Recursive types (3/7)

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

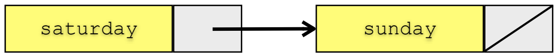
```
Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x , .. , y ]" := (cons x .. (cons y nil) ..).

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

Therefore 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 => O
  | d :: l' => S (length l')
  end.

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

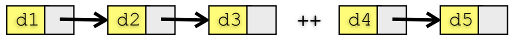
The decreasing argument is precised as hint for termination.

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.

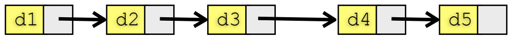
```
Fixpoint app (l1 l2 : daylist) {struct l1} : daylist :=
  match l1 with
  | nil => l2
  | d :: t => d :: (app t l2)
  end.

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.
```

Recursive types (7/7)

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

- Coq commands / keywords:
  - `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
| Node : nat -> natBinTree -> natBinTree -> natBinTree.
```

Abstract Syntax Trees for terms.

```
Inductive term : Set :=
| Zero : term
| One : term
| Plus : term -> term -> term
| Mult : term -> term -> term.
```

Other recursive datatypes (2/2)

Counting leaves and nodes in binary trees.

```
Fixpoint 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 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.
```