5th Asian-Pacific Summer School on Formal Methods

August 5-10, 2013, Tsinghua University, Beijing, China

Inductive data types (II)

jean-jacques.levy@inria.fr

August 5, 2013

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

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

Plan

- easy proofs by simplification and reflexivity
- recursive types
- recursive definitions
- structural induction
- example1: lists
- example2: trees

Recursive types

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH

Recursive types (1/6)


```
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 types (2/6)

... 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) ..).

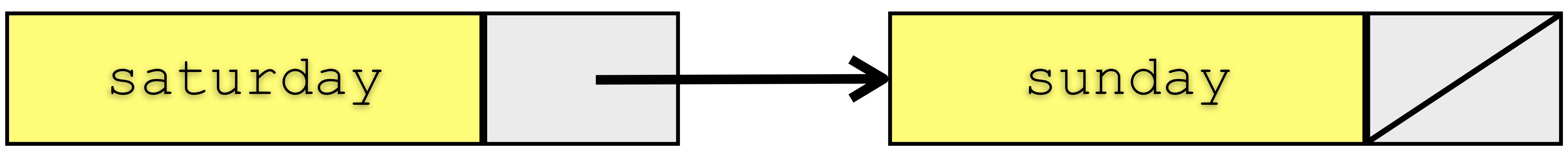
Notation "x + y" := (plus x y)
                    (at level 50, left associativity).
```

Therefore weekend_days can be also written:

```
Definition weekend_days := saturday :: sunday :: nil.
```

or

```
Definition weekend_days := [saturday, sunday].
```



Recursive types (3/6)

... 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.

Recursive types (4/6)

... with recursive definitions of functions.

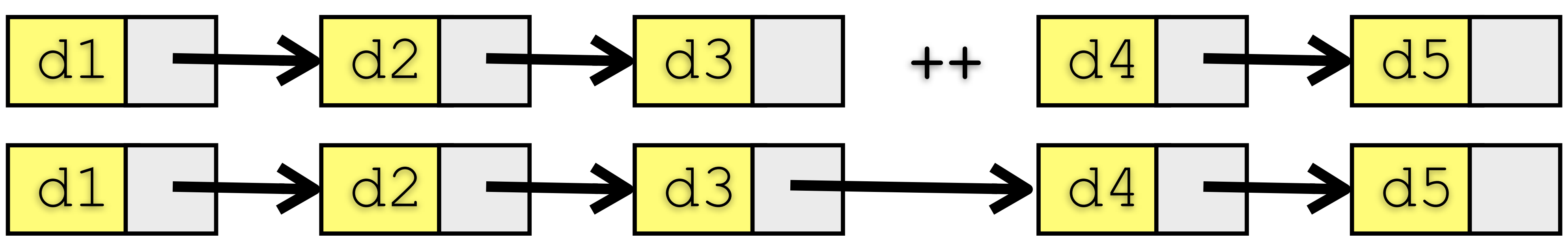
```
Fixpoint app (l1 l2 : daylist) {struct l1} : daylist :=
  match l1 with
  | nil => l2
  | d :: t => d :: (app t l2)
  end.

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 (5/6)

... 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 (6/6)

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.

Induction

Recursive types / structural induction (1/9)

Let us go back to the definition of list of days:

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

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

```
daylist_ind : forall P : daylist -> Prop,
  P nil ->

  (forall (d: day) (l1: daylist), P l1 -> P (cons d l1)) ->

  forall l : daylist, P l
```

Recursive types / structural induction (2/9)

For any $P : daylist \to Prop$, to prove that the theorem

```
forall l : daylist, P l
```

holds, it is sufficient to:

- Prove that the property holds for the base case:
  - `(P nil)`
- Prove that the property is transmitted inductively:
  - ```
    forall (d : day) (l1 : daylist),
      P l1 -> P (d :: l1)
    ```

The type `daylist` is the smallest type containing `nil` and closed under `cons`.
    ```

Recursive types / structural induction (3/9)

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

- Program by recursion (`Fixpoint`)
- Prove by induction (`induction`)

Example: append on lists.

```
Fixpoint app (l1 l2 : daylist) {struct l1} : daylist :=
  match l1 with
  | nil => l2
  | d1 :: l1' => d1 :: (app l1' l2)
  end.
```

Recursive types / structural induction (4/9)

Associativity of append on lists.

```
Theorem ass_app : forall l1 l2 l3 : daylist,
  l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.
Proof.
  intros l1 l2 l3. induction l1 as [ | d1 l1' IHl1'].
```

[ ] ++ l2 ++ l3 = ([ ] ++ l2) ++ l3

```
  - reflexivity.
```

d1 : day
l1' : daylist
l2 : daylist
l3 : daylist
IHl1' : l1' ++ l2 ++ l3 = (l1' ++ l2) ++ l3
============================
(d1 :: l1') ++ l2 ++ l3 = ((d1 :: l1') ++ l2) ++ l3

```
  - simpl. rewrite IHl1'. reflexivity.
Qed.
```

Recursive types / structural induction (5/9)

Length of appended lists.

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

```
Theorem app_length : forall l1 l2 : daylist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| d1 l1' IHl1'].
  - reflexivity.
  - simpl. rewrite IHl1'. reflexivity.
Qed.
```

Recursive types / structural induction (6/9)

Induction on natural numbers.

```
Lemma n_plus_zero : forall n:nat, n + 0 = n.
Proof.
  intros n. induction n as [| n' IH].
  - reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.

Lemma n_plus_succ : forall n m :nat, n + S m = S (n + m).
Proof.
  intros n m. induction n as [| n' IH].
  - reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.
```

Exercice 11 Show associativity and commutativity of +.

Recursive types / structural induction (7/9)

Exercice 12 Show

```
length (alternate l1 l2) = (length l1) + (length l2).
```

where

```
Fixpoint alternate (l1 l2 : daylist) {struct l1} : daylist :=
  match l1 with
  | [ ] => l2
  | v1 :: l1' => match l2 with
    | [ ] => l1
    | v2 :: l2' => v1 :: v2 :: alternate l1' l2'
    end
  end.
```

Recursive types / structural induction (8/9)

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.
```

Recursive types / structural induction (9/9)

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.
```

Exercice 13 Show

```
Lemma leaves_and_nodes : forall t : natBinTree,
  count_leaves t = 1 + count_nodes t.
```