5th Asian-Pacific Summer School on Formal Methods

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

Polymorphic types

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)

Enumerated types

CENTRE DE RECHERCHE COMMUN INRIA MICROSOFT RESEARCH

Plan

- easy proofs by simplification and reflexivity
- higher-order functions
- data types
- notation in Coq
- enumerated sets
- pattern-matching on constructors

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.

Inductive declarations

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

Enumeratives types (1/5)

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

```
Inductive bool : Set := true : bool | false : bool.
```

Set is deprecated. Now use Type.

```
Inductive color : Type := black : color | white : color.
```

Inductive declarations

Inductive types in *Coq* can be seen as the generalization of similar type constructions in more common programming languages.

They are in fact an extremely rich way of defining data-types, operators, connectives, specifications,...

They are at the core of powerful programming and reasoning techniques.

Enumeratives types (2/5)

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

A new enumerated type:

```
Inductive day : Type :=
| monday | tuesday | wednesday |
| thursday | friday | saturday | sunday : day.
```

Enumeratives types (3/5)

Inspect the enumerated type inhabitants and assign values:

```
Definition negb (b : bool) :=
  match b with true => false | false => true end.
```

Enumeratives types (4/5)

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with true => b2 | false => false  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with true => true | false => b2  end.
```

Enumeratives types (5/5)

Exercice Give definitions of predicates `work_day` and `weekend_day`.

Exercice Give definitions of predicates `black_if_workday` and white for weekends.

Polymorphic lists (1/5)

lists of any type X.

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.
```

Exercice 14 Check *list*, *nil*, *cons*.
Exercice 15 Check *cons nat 1 (cons nat 2 (nil nat))*.

```
Fixpoint length (X:Type) (l:list X) {struct l} : nat :=
  match l with
  | nil => 0
  | cons h t => S (length X t)
  end.
```

```
Example test_length1 :
    length nat (cons nat 1 (cons nat 2 (nil nat))) = 2.
Proof. reflexivity. Qed.
```

No need to put X as argument of *nil* and *cons* in matching.

Polymorphic lists (2/5)

```
Fixpoint app (X:Type) (l1 l2 : list X) {struct l1}
                : (list X) :=
  match l1 with
  | nil => l2
  | cons h t => cons X h (app X t l2)
  end.
```

Exercice 16 Associativity of append. Etc..

```
Fixpoint rev (X:Type) (l:list X) {struct l} : list X :=
  match l with
  | nil => nil X
  | cons h t => app X (rev X t) (cons X h (nil X))
  end.
```

Polymorphic lists (3/5)

```
Definition daylist := list day.
```

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

Synthetizing arguments (1/3)

```
Fixpoint length (X:Type) (l:list X) {struct l} : nat :=
  match l with
  | nil => 0
  | cons h t => S (length _ t)
  end.
```

```
Example test_length2 :
    length _ (cons _ 1 (cons _ 2 (nil _))) = 2.
Proof. reflexivity. Qed.
```

Synthetizing arguments (2/3)

```
Implicit Arguments nil [X].
Implicit Arguments cons [X].
Implicit Arguments length [X].
Implicit Arguments app [X].
...
```

or simply with argument in braces at function definition.

```
Fixpoint length {X:Type} (l:list X) {struct l} : nat :=
  match l with
  | nil => 0
  | cons h t => S (length t)
  end.
```

```
Example test_length3 :
    length (cons 1 (cons 2 (nil))) = 2.
Proof. reflexivity. Qed.
```

`@length is notation for function with all arguments.`

Synthetizing arguments (3/3)

Also decreasing argument is implicit when clear from definition.

```
Fixpoint length {X:Type} (l:list X) : nat :=
  match l with
  | nil => 0
  | cons h t => S (length t)
  end.
```

```
Fixpoint app {X : Type} (l1 l2 : list X) : (list X) :=
  match l1 with
  | nil => l2
  | cons h t => cons h (app t l2)
  end.
```

Exercice 17 Write definition of *rev* with implicit arguments.

Polymorphic lists (4/5)

Let iterative reverse be:

```
Fixpoint irev {X: Type} (l1 l2: list X) : list X :=
  match l1 with
  | [ ] => l2
  | v1 :: l1' => irev l1' (v1 :: l2)
  end.
```

Exercice 18 Show for any lists ℓ_1, ℓ_2, ℓ_3:

$\ell_1 ++ (\ell_2 ++ \ell_3) = (\ell_1 ++ \ell_2) ++ \ell_3$
$\text{length}(\ell_1 ++ \ell_2) = (\text{length}\,\ell_1) + (\text{length}\,\ell_2)$
$\text{rev}\ \ell_1 = \text{irev}\ \ell_1\,[\,]$
$\ell ++ [\,] = \ell$
$\text{rev}(\ell_1 ++ \ell_2) = (\text{rev}\,\ell_2) ++ (\text{rev}\,\ell_1)$
$\text{rev}(\text{rev}\,\ell) = \ell$
$\ell = \text{rev}\ \ell' \ \Rightarrow\ \ell' = \text{rev}\ \ell$

Polymorphic binary trees (1/2)

```
Inductive binTree (X : Type) :=
| leaf : X -> binTree X
| node : X -> binTree X -> binTree X -> binTree X.

Fixpoint count_leaves {X: Type} (t : binTree X) :=
  match t with
  | leaf _ => 1
  | node _ t1 t2 => (count_leaves t1) + (count_leaves t2)
  end.
```

Polymorphic binary trees (2/2)

```
Lemma height_le_size : forall (X: Type) (t : binTree X),
  height t <= size t.
Proof.
intros X t. induction t as [| x t1 IHt1 t2 IHt2].
- reflexivity.
- simpl. apply Le.le_n_S.
  apply Max.max_case.
  + apply (Le.le_trans _ (size t1) _).
    apply IHt1. apply Plus.le_plus_l.
  + apply (Le.le_trans _ (size t2) _).
    apply IHt2. apply Plus.le_plus_r.
Qed.
```

Higher order functions

```
Fixpoint map X Y: Type (f : X->Y) (l : list X) struct l: list Y :=
  match l with
  | [ ] => [ ]
  | x :: l' => (f x) :: map f l'
  end.

Example map_negb : map negb [true, false] = [false, true].
Example map_next_weekday :
    map next_weekday [monday, friday] = [tuesday, monday].
```

Exercice 19 Show
map f (rev ℓ) = rev(map f ℓ)
$map\ f\ (\ell_1 \text{ ++ } \ell_2) = (\text{map } f\ \ell_1) \text{ ++ } (\text{map } f\ \ell_2)$

Polymorphic Option and Product

A polymorphic non recursive option type:

```
Inductive option (X : Type) : Type :=
  Some : X -> option X | None : option X
```

Use it for default value:

```
Fixpoint last {X : Type} (l : list X) : option X :=
  match l with
    | [ ] => None
    | v :: nil => Some v
    | _ :: l' => last l'
  end.
```

We also define polymorphic product.

```
Inductive prod {X Y : Type} : Type :=
    pair : X -> Y -> prod X Y
```

The notation `X * Y` denotes `(prod X Y)`.
The notation `(x, y)` denotes `(pair x y)` (implicit argument).

Functions (1)

jean-jacques.levy@inria.fr
5th Asian-Pacific Summer School on Formal Methods
Tsinghua Univ., Beijing
August 5, 2013

http://jeanjacqueslevy.net/courses/13eci

5th Asian-Pacific Summer School on Formal Methods

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

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