Pauta

21/07/2018

- Participantes;
- Comunicação;
- Repositório;
- Datas, local e periodicidade;
- Referências;
- Dinâmica e metodologia;
- Objetivos;
- Exemplo motivacional (??)

Referências

- Type Theory and Functional Programming
  Simon Thompson
  https://www.cs.kent.ac.uk/people/staff/sjt/TTFP/ttfp.pdf

- Interactive Theorem Proving and Program Development
  Yves Bertot and Pierre Castéran
  https://www.labri.fr/perso/casteran/CoqArt/

- Software Foundations
  Vol. 1 – Logical Foundations
  Benjamin Pierce et al.
  https://softwarefoundations.cis.upenn.edu/

Pré-requisitos

- Muita teoria por trás;
  - ✓ Lógica
  - ✓ Dedução Natural
  - ✓ Cálculo Lambda
  - ✓ Teoria de Tipos
  - ✓ ...
- Linguagens:
  - ✓ Vernacular;
  - ✓ Gallina;
  - ✓ Sintaxe;
  - ✓ Semântica
- CoqIDE;
- Inglês (!!)

Exemplo

- Muitos detalhes;
- Não se preocupem em entender tudo;
- Busquem apenas uma intuição inicial do que está sendo feito e como está sendo feito;
- Mais importante é ter uma visão geral da dinâmica e do tipo de trabalho envolvido;
- A plena compreensão virá depois, com o tempo e a prática.

Script Coq

- Texto corrido;
- Processado de cima para baixo, esquerda para a direita;
- Mensagens de erros e interação com o usuário;
- Definições (indutivas e não-indutivas);
- Funções (recursivas e não-recursivas);
- Lemas e teoremas;
  (proposições provadas de forma interativa usando um conjunto de táticas e regras de inferência; as provas são criadas indiretamente)

Script Coq

- Novos nomes;
- Utilização nas etapas seguintes;
  (lema A é usado para provar B, que por sua vez é usado para provar C e assim por diante)
- Computação e dedução;
- Lema ou teorema final;
- Provas completas;
  - ✓ Contexto;
  - ✓ Indução;
  - ✓ O script não é a prova!
- Extração de código.

Motivação

- Construir um programa certificado que ordena listas de números inteiros.

- Lista?
- Lista ordenada?
- Qual seria a especificação deste programa?

- Uma vez especificado, construímos a prova;
- Da prova, extraímos o programa certificado.

- Série de definições (algumas indutivas outras não) e lemas.

Número natural

- Um tipo de dados definido de maneira indutiva.
- Dois construtores apenas.

```
Inductive nat: Type :=
 | O : nat
 | S : nat -> nat.
```

Número natural

- Um tipo de dados definido de maneira indutiva.
- Dois construtores apenas.

```
O                0
S O              1
S (S O)          2
S (S (S O))      3
...
```

Lista

- Um tipo de dados definido de maneira indutiva;
- Parametrizado em função do tipo do elemento (A).

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

Lista

```
nil
ou
[]

cons 3 nil
ou apenas
3 :: nil
ou ainda
[3]

cons (3 (cons (4 (cons 5 nil))))
ou apenas
3 :: 4 :: 5 :: nil
ou ainda
[3;4;5]
```

Lista ordenada

- Uma coleção infinita de proposições definida de maneira indutiva;
- Lista de números inteiros (Z).

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1 : forall z:Z, sorted (z :: nil)
| sorted2 : forall z1 z2:Z,
            forall l:list Z,
            z1 <= z2 -> sorted (z2 :: l) ->
            sorted (z1 :: z2 :: l).
```

Lista ordenada

```
sorted nil
Pelo construtor sorted0

sorted1 (3 :: nil)
Pelo condtrutor sorted1

sorted2 (2 :: 3 :: nil)
Pelos construtores sorted1 e depois sorted2

sorted2 (1 :: 2 :: 3 :: nil)
Pelos construtores sorted 1, sorted2 e sorted2.
```

Lema

```
Lemma sorted_example:
sorted (2 :: 3 :: 5 :: 7 :: nil).
Proof.
apply sorted2.
- omega.
- apply sorted2.
  + omega.
  + apply sorted2.
    * omega.
    * apply sorted1.
Qed.
```

Teorema

```
Theorem sorted_inv :
forall z:Z,
forall l:list Z,
sorted (z :: l) -> sorted l.
Proof.
intros z l H.
inversion H.
- apply sorted0.
- exact H3.
Qed.
```

Função recursiva

- Calcula e retorna o número de ocorrências de um número numa lista de números.

```
Fixpoint nb_occ (z:Z) (l:list Z): nat:=
match l with
| nil => 0%nat
| (z' :: l') =>
  match Z_eq_dec z z' with
  | left _ => S (nb_occ z l')
  | right _ => nb_occ z l'
  end
end.
```

Definição

- Proposição parametrizada definida de maneira não indutiva.

```
Definition permutation (l l':list Z) : Prop :=
forall z:Z, nb_occ z l = nb_occ z l'.
```

Função recursiva

- Insere um novo número numa lista ordenada de números de tal forma que a lista resultante continue ordenada.

```
Fixpoint insert (z:Z) (l:list Z): list Z :=
match l with
| nil => z :: nil
| cons a l' =>
  match Z_le_gt_dec z a with
  | left _ =>  z :: a :: l'
  | right _ => a :: (insert z l')
  end
end.
```

Objetivo final

- Prova a proposição abaixo, que garante existir uma lista ordenada para qualquer lista que se considere.
- O programa certificado é extraído da prova desta proposição, juntamente com algumas outras.

```
Lemma sort_correct:
forall l: list Z,
exists l': list Z,
permutation l l' /\ sorted l'.
```

O script da prova

```
Proof.
induction l.
- exists nil.
  split.
  + apply permutation_refl.
  + apply sorted0.
- destruct IHl as [l' [H1 H2]].
  exists (insert a l').
  split.
  + apply permutation_trans with (l2:= a :: l').
    * apply permutation_cons.
      exact H1.
    * apply insert_permutation.
  + apply insert_sorted.
    exact H2.
Qed.
```

A prova

```
sort_correct =
fun l : list Z =>
list_ind
  (fun l0 : list Z => exists l' : list Z, permutation l0 l' /\ sorted l')
  (ex_intro (fun l' : list Z => permutation nil l' /\ sorted l') nil
     (conj (permutation_refl nil) sorted0))
  (fun (a : Z) (l0 : list Z)
     (IHl : exists l' : list Z, permutation l0 l' /\ sorted l') =>
   match IHl with
   | ex_intro _ l' (conj H1 H2) =>
       ex_intro (fun l'0 : list Z => permutation (a :: l0) l'0 /\ sorted l'0)
         (insert a l')
         (conj
            (permutation_trans (a :: l0) (a :: l') (insert a l')
               (permutation_cons a l0 l' H1) (insert_permutation l' a))
            (insert_sorted l' a H2))
   end) l
     : forall l : list Z, exists l' : list Z, permutation l l' /\ sorted l'
```

Código ocaml extraído

```
type __ = Obj.t
let __ = let rec f _ = Obj.repr f in Obj.repr f
type 'a list =
| Nil
| Cons of 'a * 'a list
type comparison =
| Eq
| Lt
| Gt
(** val compOpp : comparison -> comparison **)
let compOpp = function
| Eq -> Eq
| Lt -> Gt
| Gt -> Lt
type sumbool =
| Left
| Right
type positive =
| XI of positive
| XO of positive
| XH
type z =
| Z0
| Zpos of positive
| Zneg of positive
module Pos =
 struct
  (** val compare_cont : comparison -> positive -> positive -> comparison **)
  let rec compare_cont r x y =
    match x with
    | XI p ->
      (match y with
       | XI q -> compare_cont r p q
       | XO q -> compare_cont Gt p q
       | XH -> Gt)
    | XO p ->
      (match y with
       | XI q -> compare_cont Lt p q
       | XO q -> compare_cont r p q
       | XH -> Gt)
    | XH ->
      (match y with
       | XH -> r
       | _ -> Lt)
  (** val compare : positive -> positive -> comparison **)
  let compare =
    compare_cont Eq
 end
module Z =
 struct
  (** val compare : z -> z -> comparison **)
  let compare x y =
    match x with
    | Z0 ->
      (match y with
       | Z0 -> Eq
       | Zpos _ -> Lt
       | Zneg _ -> Gt)
    | Zpos x' ->
      (match y with
       | Zpos y' -> Pos.compare x' y'
       | _ -> Gt)
    | Zneg x' ->
      (match y with
       | Zneg y' -> compOpp (Pos.compare x' y')
       | _ -> Lt)
 end
(** val z_le_dec : z -> z -> sumbool **)
let z_le_dec x y =
  match Z.compare x y with
  | Gt -> Right
  | _ -> Left
(** val z_le_gt_dec : z -> z -> sumbool **)
let z_le_gt_dec x y =
  z_le_dec x y
(** val insert : z -> z list -> z list **)
let rec insert z0 = function
| Nil -> Cons (z0, Nil)
| Cons (a, l') ->
  (match z_le_gt_dec z0 a with
   | Left -> Cons (z0, (Cons (a, l')))
   | Right -> Cons (a, (insert z0 l')))
(** val sort_correct : __ **)
let sort_correct =
  __
```