

Programação com tipos dependentes

Introdução ao assistente de provas Coq

Rodrigo Ribeiro

Tipos dependentes

- ▶ Permitem combinar programas e suas respectivas provas de correção.
- ▶ Tipo subset

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=  
  exist : forall x : A, P x -> sig P
```

Tipo subset

- ▶ Tipo `sig A P` representado por `{x : A | P x}`.
 - ▶ Conjunto de valores `x : A` tais que `P x`.

Definition `plus_cert`
: forall (n m : nat), {r : nat | Plus n m r}.

Tipo subset e tática refine.

```
refine (fix plus_cert (n m : nat) : {r | Plus n m r} :=  
  match n return {r | Plus n m r} with  
  | 0      => exist _ m _  
  | S n1 =>  
    match plus_cert n1 m with  
    | exist _ r1 _ => exist _ (S r1) _  
    end  
  end)  
; clear plus_cert ; auto.  
Defined.
```

Exemplo: Predecessor de um número natural

```
Definition pred_cert
  : forall (n : nat), n > 0 -> {r | n = 1 + r}.
refine (fun n =>
  match n return n > 0 -> {r | n = 1 + r} with
  | 0 => fun _ => False_rec _ _
  | S n' => fun _ => exist _ n' _
  end) ; omega.
```

Defined.

Notações para uso de tipo subset

Notation "!" := (False_rec _ _).

Notation "[e]" := (exist _ e _).

Exemplo usando notações

```
Definition pred_cert1
  : forall (n : nat), n > 0 -> {r | n = 1 + r}.
  refine (fun n =>
    match n return n > 0 -> {r | n = 1 + r} with
    | 0 => fun _ => !
    | S n' => fun _ => [ n' ]
    end) ; omega.
```

Defined.

Tipo sumbool

- ▶ Útil para definir proposições decidíveis
 - ▶ O tipo `sumbool A B` é representado por $A + B$.

```
Inductive sumbool (A B : Prop) : Set :=  
| left  : A -> {A} + {B}  
| right : B -> {A} + {B}
```


Notações para o tipo sumbool

Notation "'Yes'" := (left _ _).

Notation "'No'" := (right _ _).

Notation "'Reduce' x" :=
 (if x then Yes else No) (at level 50).

Exemplo

Definition eq_nat_dec

: forall (n m : nat), {n = m} + {n <> m}.

refine (fix eq_nat n m : {n = m} + {n <> m} :=

match n , m with

| 0 , 0 => Yes

| S n' , S m' => Reduce (eq_nat n' m')

| _ , _ => No

end) ; clear eq_nat ; congruence.

Defined.

Estudo de caso: Busca em finite map.

Variable Key : Type.

Variable Value : Type.

Variable eqKeyDec :

forall (x y : Key), {x = y} + {x <> y}.

Inductive Map : Type :=

| nil : Map

| cons : Key -> Value -> Map -> Map.

Estudo de caso: Busca em finite map.

- ▶ Pertinência em finite maps, indutivamente.

Inductive MapsTo

```
    : Key -> Value -> Map -> Prop :=  
| Here  : forall k v m,  
    MapsTo k v (cons k v m)  
| There : forall k v m k' v',  
    k <> k' ->  
    MapsTo k v m ->  
    MapsTo k v (cons k' v' m).
```

Hint Constructors MapsTo.

Estudo de caso: Busca em finite map.

- Tipo da função de busca

```
Definition lookupMap
  : forall (k : Key)(m : Map),
    {v | MapsTo k v m} +
    {forall v, ~ MapsTo k v m}.
```

Estudo de caso: Busca em finite map.

```
refine (fix look k m :  
  {v | MapsTo k v m} + {forall v, ~ MapsTo k v m} :=  
    match m return  
      {v | MapsTo k v m} + {forall v, ~ MapsTo k v m} with  
      | nil => !!  
      | cons k' v' m' =>  
        match eqKeyDec k k' with  
        | Yes => [|| v' ||]  
        | No  =>  
          match look k m' with  
          | !! => !!  
          | [|| v ||] => [|| v ||]  
          end  
        end  
    end)  
end) ;
```

Estudo de caso: Busca em finite map.

- Tática para finalizar a definição de lookup

```
clear look ; subst ;
try (repeat
  (match goal with
    | [H : MapsTo _ _ nil |- _] => inverts H
    | [H : MapsTo _ _ (cons _ _ _) |- _] =>
      inverts H
    | [|- forall x, ~ _ ] => unfold not ; intros
    | [ H : forall x, ~ (MapsTo _ _ _)
      , H1 : MapsTo _ _ _ |- _] =>
      apply H in H1
  end)) ; auto.
```

Vectors

```
Inductive vector (A : Set) : nat -> Type :=  
| vnil  
  : vector A 0  
| vcons  
  : forall n, A ->  
    vector A n -> vector A (S n).
```


Exemplo: concatenação

```
Fixpoint app {A : Set}{n1 n2}
(ls1 : vector A n1)
(ls2 : vector A n2) : vector A (n1 + n2) :=
  match ls1 with
  | vnil _ => ls2
  | vcons _ _ x ls1' => vcons _ _ x (app ls1' ls2)
end.
```

Exemplo: função head

```
Definition vhead {A : Set}{n}
  (v : vector A (S n)) : A :=
  match v with
  | vcons _ _ x _ => x
  end.
```

Indexando vectors

- Vectors permitem a definição de uma função segura para indexar vectors. Esta utiliza o tipo `fin`

```
Inductive fin : nat -> Set :=  
| fzero : forall {n}, fin (S n)  
| fsucc  : forall {n}, fin n -> fin (S n).
```

O tipo `fin n`

- ▶ O tipo `fin 0` não é habitado: equivalente a `False`.
- ▶ O tipo `fin n` possui *exatamente* `n` elementos.
 - ▶ O tipo `fin 1`, possui exatamente um valor `fzero : fin 1`.
 - ▶ O tipo `fin 2` possui dois a valores, a saber: `fzero : fin 2` e `fsucc fzero : fin 2`

Indexando vectors — parte 1

```
Fixpoint get {A}{n}(ls : vector A n) : fin n -> A :=
  match ls with
  | vnil _ => fun idx =>
    match idx in fin n' return (match n' with
                                | 0 => A
                                | S _ => unit
                                end) with
    | fzero => tt
    | fsucc _ => tt
  end
  ...
```

Indexando vectors — parte 2

```
| vcons _ _ x ls' => fun idx =>  
match idx in fin n'  
  return (fin (pred n') -> A) -> A with  
    | fzero => fun _ => x  
    | fsucc idx' => fun get_ls' =>  
      get_ls' idx'  
  end (get ls')  
end.
```

Indexando vectors

- ▶ O tipo de `get` garante:
 - ▶ Essa função não pode ser aplicada a `vectors` vazios.
 - ▶ Essa função acessa somente posições válidas do `vector`.