

Práctico 6

Prueba de Programas

- Programas Funcionales -

Objetivos: Extraer programas funcionales a partir de pruebas. Probar la corrección de programas con respecto a una especificación: terminación de la recursión y corrección propiamente dicha. Especificación de programas.

Principales tácticas a utilizar en estos ejercicios:

- `Extraction Language ...`
- `Extraction "..."` IDlema (y variantes)
- **Functional Scheme, Function, functional induction.**

Principales bibliotecas a consultar

- `[-] theories\RELATIONS\WELLFOUNDED\` y en particular `Inverse_Image.v`.
- `[-] theories\ARITH\` y en particular `Wf_nat.v`.

Principales herramientas automáticas

- `Hint`
- `Auto`
- `Omega`

Ejercicio 6.1.

1. Demuestre en Coq el siguiente lema que especifica la función predecesor para números naturales:
`Lemma predspeg : forall n : nat, {m : nat | n = 0 /\ m = 0 \/ n = S m}.`
2. Realice la siguiente secuencia de pasos para extraer su programa Coq en un programa Haskell.
`Extraction Language Haskell.`
`Extraction "predecesor" predspeg.`
3. Inspeccione el archivo `predecesor.hs` para ver el código extraído. Puede cargarlo llamando al compilador de Haskell.

Ejercicio 6.2.

1. Considere las definiciones de árbol binario y espejo del práctico 4. Demuestre que para todo árbol binario existe otro que es su espejo, o sea,
`Lemma MirrorC: forall (A:Set) (t:bintree A),
 {t':bintree A | (mirror A t t')}.`
2. Redemuestre el lema anterior usando (verificando) la función `inverse` del práctico 4 (la

Construcción Formal de Programas en Teoría de Tipos

cual, dado un árbol binario construye otro que es su espejo). Considere la declaración:
Hint Constructors mirror, y analice la táctica “functional induction”.

3. Extraiga su programa Coq en un program Haskell llamado `mirror_function.hs` e inspeccione el archivo para ver el código extraído.

Ejercicio 6.3.

1. Considere la siguiente simplificación de los tipos del ejercicio 5.5 del práctico anterior.

```
Definition Value := bool.
```

```
Inductive BoolExpr : Set :=  
  | bbool : bool -> BoolExpr  
  | band : BoolExpr -> BoolExpr -> BoolExpr  
  | bnot : BoolExpr -> BoolExpr.
```

```
Inductive BEval : BoolExpr -> Value -> Prop :=  
  | ebool : forall b : bool, BEval (bbool b) (b:Value)  
  | eandl : forall e1 e2 : BoolExpr, BEval e1 false -> BEval (band e1 e2) false  
  | eandr : forall e1 e2 : BoolExpr, BEval e2 false -> BEval (band e1 e2) false  
  | eandrl : forall e1 e2 : BoolExpr,  
    BEval e1 true -> BEval e2 true -> BEval (band e1 e2) true  
  | enott : forall e : BoolExpr, BEval e true -> BEval (bnot e) false  
  | enotf : forall e : BoolExpr, BEval e false -> BEval (bnot e) true.
```

y los siguientes programas de evaluación de expresiones.

```
Fixpoint beval1 (e : BoolExpr) : Value :=  
  match e with  
  | bbool b => b  
  | band e1 e2 =>  
    match beval1 e1, beval2 e2 with  
    | true, true => true  
    | _, _ => false  
    end  
  | bnot e1 => if beval1 e1 then false else true  
  end.
```

```
Fixpoint beval2 (e : BoolExpr) : Value :=  
  match e with  
  | bbool b => b  
  | band e1 e2 => match beval2 e1 with  
    | false => false  
    | _ => beval2 e2  
    end  
  | bnot e1 => if beval2 e1 then false else true  
  end.
```

Demuestre sendos lemas de corrección (`beval1C` y `beval2C`) que establezcan que los programas `beval1` y `beval2` son correctos con respecto a la especificación:

```
forall e:BoolExpr, {b:Value | (BEval e b)}.
```

2. Redemuestre los lemas poniendo en Hint los constructores de la relación `BEval`.

Construcción Formal de Programas en Teoría de Tipos

3. Extraiga de los lemas de corrección código Haskell de los evaluadores demostrados.
4. Regenera el archivo Haskell del punto anterior de forma que el tipo `bool` de Coq sea extraído como el tipo `bool` de Haskell.

Ejercicio 6.4.

Considere las siguientes definiciones que formalizan la relación de permutación entre listas:

```
Section list_perm.
```

```
Variable A:Set.
```

```
Inductive list : Set :=  
  | nil : list  
  | cons : A -> list -> list.
```

```
Fixpoint append (l1 l2 : list) {struct l1} : list :=  
  match l1 with  
  | nil => l2  
  | cons a l => cons a (append l l2)  
end.
```

```
Inductive perm : list -> list -> Prop :=  
|perm_refl: forall l, perm l l  
|perm_cons: forall a l0 l1, perm l0 l1 -> perm (cons a l0) (cons a l1)  
|perm_app: forall a l, perm (cons a l) (append l (cons a nil))  
|perm_trans: forall l1 l2 l3, perm l1 l2 -> perm l2 l3 -> perm l1 l3.
```

```
Hint Constructors perm.
```

1. Defina una función `reverse` que dada una lista retorne la lista invertida.
2. Pruebe que la función `reverse` de una lista es una implementación de la siguiente especificación:

```
Lemma Ej6_4: forall l: list, {l2: list | perm l l2}.
```

```
...
```

```
End list_perm.
```

Ejercicio 6.5.

1. Defina los predicados `Le:nat->nat->Prop` y `Gt:nat->nat->Prop` que representan las relaciones *menor o igual* y *mayor* entre números naturales respectivamente.
2. Demuestre que el orden entre números naturales es decidible probando el siguiente lema:
`Le_Gt_dec: forall n m:nat, {(Le n m)}+{(Gt n m)}`. Para ello, escriba un programa `leBool:nat->nat->bool` que “decida” si un número natural es menor o igual que otro y utilícelo junto con la táctica `functional induction` en la prueba del lema.
3. Considere la función `leBool` definida en la parte anterior. Demuestre el lema de

Construcción Formal de Programas en Teoría de Tipos

decidibilidad `le_gt_dec`: forall n m:nat, {(le n m)}+{(gt n m)} donde `le` y `gt` son las relaciones de la biblioteca Coq. Para hacer la prueba emplee la táctica `functional induction` e intente demostrar los objetivos aritméticos con la táctica `omega` (incluya previamente el módulo `Omega`).

Ejercicio 6.6.

Considere la siguiente especificación de la división euclideana vista en el curso:

```
Definition spec_res_nat_div_mod (a b:nat) (qr:nat*nat) :=  
  match qr with  
  (q,r) => (a = b*q + r) /\ r < b  
  end.
```

```
Definition nat_div_mod :  
  forall a b:nat, not(b=0) -> {qr:nat*nat | spec_res_nat_div_mod a b qr}.
```

Derive a partir de la especificación anterior un algoritmo para la división.

Sugerencia: considere en la prueba la lógica de la siguiente solución (con $b > 0$):

- `0 divmod b = (0,0)`
- `(n+1) divmod b = let (q,r) = n divmod b
 in if r < b-1
 then (q,r+1)
 else (q+1,0)`

Incorpore al contexto los siguientes módulos:

```
Require Import Omega.  
Require Import DecBool.  
Require Import Compare_dec.  
Require Import Plus.  
Require Import Mult.
```

Ejercicio 6.7.

Considere las siguientes definiciones que permiten formalizar una relación de subárbol entre árboles binarios.

```
Inductive tree (A:Set) : Set :=  
  | leaf : tree A  
  | node : A -> tree A -> tree A -> tree A.
```

```
Inductive tree_sub (A:Set) (t:tree A) : tree A -> Prop :=  
  | tree_sub1 : forall (t':tree A) (x:A), tree_sub A t (node A x t t')  
  | tree_sub2 : forall (t':tree A) (x:A), tree_sub A t (node A x t' t).
```

Pruebe que la relación `tree_sub` es un orden bien fundado.

Construcción Formal de Programas en Teoría de Tipos

`Theorem well_founded_tree_sub : forall A:Set, well_founded (tree_sub A).`

Ejercicio 6.8.

Considere los tipos `Value`, `BoolExpr` y `BEval` definidos en el ejercicio 3.

1. Defina un orden bien fundado `elt` (*Expressions Less Than*) que justifique la terminación de los programas que evalúan expresiones (de forma ansiosa y perezosa) definidos en el ejercicio 3. Defina el orden a partir de una función `size` de tipo `BoolExpr->nat`, como sigue:

`Definition elt (e1 e2 : BoolExpr) := size e1 < size e2.`

2. Demuestre que el orden `elt` es bien fundado. Sugerencia: utilice los módulos `Wf_nat` e `Inverse_Image`.

Ejercicio 6.9.

Considere lista de naturales y la función `insert_sort` del práctico 4. Demuestre que dicha función es una implementación correcta de la siguiente especificación:

`SORT: forall l:(list nat), {s:(list nat) |
 (sorted nat le s) /\ (perm nat l s)}, donde:`

`sorted: forall A:Set, (R:A->A->Prop)->(l:list A)->Prop`, es el predicado definido en el práctico 4,

`le:nat->nat->Prop` es el orden \leq entre números naturales, y

`perm: forall A:Set, (list A)->(list A)->Prop` es la relación de permutación entre listas.

Nota: considere la variante de la definición de permutaciones entre listas del ejercicio 4 que sigue:

Sustituir el constructor:

`|perm_app: forall a l, perm (cons a l) (append l (cons a nil))`

por el constructor:

`|p_ccons: forall a b l (perm (cons a (cons b l)) (cons b (cons a l)))`

Ejercicios a entregar:

Ver la fecha límite y los ejercicios requeridos en el sitio EVA del curso.

El archivo a entregar debe cargar correctamente en Coq. Si deja ejercicios sin resolver, debe delimitarlos como comentarios: (... *).*

Al inicio del archivo deben estar los datos de cada integrante; se admiten entregas individuales o de a dos estudiantes.

Construcción Formal de Programas en Teoría de Tipos

Usar la plantilla publicada junto con el práctico para el desarrollo de los ejercicios requeridos; no es necesario entregar los ejercicios no solicitados.