6. Verificación de programas funcionales

Especificación de programas

- $\forall i \in$ **Input.** ψ**(i)**$\rightarrow \exists o \in$ **Output.** φ**(i,o)**

  donde Input y Output son **tipos de datos,** ψ es una **proposición** que expresa cierta condición que debe cumplir el valor de entrada y φ es la **proposición** que expresa la relación que debe cumplirse entre el valor de entrada y el valor de salida

En general:

- $\forall i_1 \in \mathbf{Input_1} \ldots \forall i_n \in \mathbf{Input_n}.\ \psi(i_{1\ldots}\, i_n) \rightarrow \exists o \in \mathbf{Output}.\ \varphi(i_1,\ldots,i_n,o)$

Ejemplo:

Para todos $a,b \in$ **N** tq. $b\neq 0$ existen q y r tq. $a=b.q+r$ y $r < b$.

$(\forall a,b \in \mathbf{N})\ \ b\neq 0 \rightarrow \exists\, (q,r) \in \mathbf{N}\times\mathbf{N}.\ a=b.q+r \ \wedge\ r < b$

Especificaciones en Coq

En Coq:

forall **(i_1:Input$_1$)... (i_n:Input$_n$), ψ($i_{1\ldots}$ i_n) $\rightarrow$ {o:Output | φ(i_1,...,i_n,o) }**

Ejemplos:

- **forall (*a b:nat*), b$\neq$0 $\rightarrow$ {qr: N*N | match qr with**
  (*q,r*) => *a=b.q+r* $\wedge$ *r < b*
  end }

- ¿Especificación de la función que ordena una lista?

  forall l:list, { l': list | (sorted l') $\wedge$ (perm l l') }

Extracción de programas

Si la prueba del existencial

$$\forall i \in \text{Input}.\ \psi(i) \rightarrow \{o \in \text{Output} \mid \varphi(i,o)\}$$

es ***constructiva***, el programa que calcula el resultado a partir de la entrada se encuentra necesariamente ***embebido*** en la prueba.

- *Qué diferencia el programa de la prueba?*
  - La ***prueba*** es el ***programa*** más la ***información lógica*** necesaria para demostrar la especificación
  - Para obtener el programa hay que ***extraer*** de la prueba la información computacional y ***olvidar*** la información lógica

Extracción de programas

Información computacional : objetos que viven en **Set**

Información lógica : objetos que viven en **Prop**

Mecanismo de extracción :

- Recorre el término de prueba recuperando los datos que viven en Set, olvidando los que viven en Prop y manteniendo la estructura.
- Originalmente se extrae hacia Fw, un lenguaje de programación no dependiente.
- Las dependencias en los objetos computacionales se olvidan.

[Paulin-Mohring 89]

Extracción de programas

Dada una prueba

$p : \forall i \in Input.\ \psi(i) \rightarrow \{ o \in Output \mid \varphi(i,o) \}$

el *contenido computacional* de p será una función

$f_p : Input \rightarrow Output$

Para ello deberá ocurrir que :

Input : Set

Output : Set

$\psi(i)$, $\varphi(i,o)$: Prop

Observar que los sorts de los tipos que intervienen en la especificación indican al procedimiento de extracción qué recorrer y qué olvidar durante la recorrida

Existenciales en Coq
con contenido computacional

Inductive sig (A:Set)(P:A→ **Prop**) : **Set**
:= exist : forall x:A, P x → sig A P.

{ x : A | P x }

Inductive sig2 (A:Set)(P Q:A→ **Prop**) : **Set**
:= exist2 : forall x:A, P x → Q x → sig2 A P Q.

{ x : A | (P x) & (Q x) }

Inductive sigS (A:Set)(P:A→ **Set**) : **Set**
:= existS : forall x:A, P x → sigS A P.

{ x : A & (P x) }

Inductive sigS2 (A:Set)(P Q:A→ **Set**) : **Set**
:= existS2: forall x:A, P x → Q x → sigS2 A P Q.

{ x : A & (P x) & (Q x) }

Existenciales en Coq
sin contenido computacional

Inductive ex (A:Set)(P:A→ **Prop)** : **Prop**
:= ex_intro : forall x:A, P x → ex A P.

exist x : A, P x

Inductive ex2 (A:Set)(P Q:A→ **Prop)** : **Prop**
:= ex2_intro : forall x:A, P x → Q x → ex2 A P.

exist2 x : A, (P x) & (Q x)

Sumas en Coq

Inductive sumbool (A B: **Prop**) : **Set** $\quad$ **{A } + {B }**

:= left : A $\rightarrow$ (sumbool A B)

| right : B $\rightarrow$ (sumbool A B)

Inductive sumor (A:**Set**)(B: **Prop**) : **Set** $\quad$ **A + {B}**

:= left : A $\rightarrow$ (sumor A B)

| right : B $\rightarrow$ (sumor A B)

Inductive or (A B: **Prop**) : **Prop** $\quad$ **A** $\vee$ **B**

:= or_introl : A $\rightarrow$ (or A B)

| or_intror : B $\rightarrow$ (or A B)

Mecanismo de extracción en Coq

- **Extracción del programa**

  Extraction *id*

  Recursive Extraction id_1 ... Id_n

  Extraction *“file”* $_{...,}$

  Extraction *Library ...*

- **Extracción a un lenguaje de programación** (Ocaml, Haskell, Scheme, Toplevel):

  Extraction Language language

  Extract Constant $id_{coq} \Rightarrow id_{leng}$

  Extract Inductive $type_{coq} \Rightarrow type_{leng}$ **[**$c1_{ml}$... cn_{leng}**]**

  `Ver capítulo 18 del manual de Coq`

Mecanismo de extracción (síntesis)

- Metodología para obtener programas correctos:
  - **especificar en Coq el programa como un lema**
  - **extraer el programa de la prueba del lema**

- Desventaja de esta metodología:
  - la estructura del programa está oculta en el término de prueba hasta el final del proceso
  - dificultad de lectura del programa final
  - dificultad para optimizar el programa

Quisiéramos *verificar programas* más que sintetizarlos

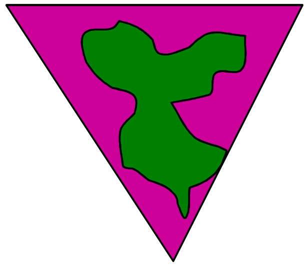
Verificación de programas

Es el proceso inverso al de extracción

P: Spec

f_P

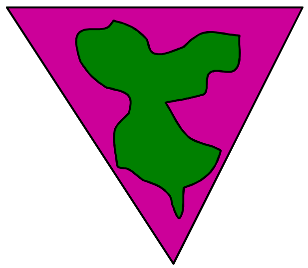
Programa



: Spec

Programa Spec

P:Spec



: Spec

Verificación de programas

- Dado el ***programa*** y la ***especificación*** hay que ***probar*** que el programa cumple con la especificación:
  - justificar la **terminación** del programa
  - probar los objetivos ligados con la **especificación** particular.
- La síntesis de esta información no es automática (a excepción de casos sencillos).
- Para simplificar el proceso de verificación los programas pueden tener anotaciones lógicas, que son directivas que guían la síntesis.

Ejemplo: División

Sean *a* y *b* $\in$ **N**, b≠0.

Calculamos *a* **div** *b* y a **mod** *b* *simultáneamente* haciendo recursión en *a*:

- 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)
```

División: especificación

- Especificación: Para todos $a,b \in \mathbf{N}$ tq. $b \neq 0$ existen q y r tq. $a=b.q+r$ y $r < b$.
- Prueba de que el programa es correcto: por inducción en *a:*

- 0 **divmod** *b* = <0,0> ➔ $0 = b.0 + 0$ y $0 < b$
- (n+1) **divmod** *b* =

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

➔ **supongamos** $n=b.q+r$ *y* $r < b.$

luego, si $r < b - 1$

entonces $n+1 = b.q+(r+1)$ *y* $r+1<b$

sino $n+1 = b.(q+1)+0$ y $0<b$

Algunas Tácticas

- **refine *term*** : esta táctica permite proveer una prueba “casi” exacta de un goal (término con “holes”, denotados _)
- **functional induction *id* $\boldsymbol{term_1}$ ... $\boldsymbol{term_n}$** : ésta es una táctica (aún experimental) que permite razonar por casos o por inducción siguiendo la definición de una función recursiva estructural

Pruebas de Terminación

- Si el programa no es recursivo el programa termina de forma trivial.
- Si el programa es recursivo hay que justificar:
  - que los llamados recursivos ***decrecen*** según cierto orden ***R***
  - que la relación ***R*** es ***bien fundada*** (o sea que no existen cadenas descendentes infinitas).
- En general, el programa debe tener como *anotación* la relación ***R*** con la cual hay que comparar los llamados recursivos

Accesibilidad

- Definición recursiva de funciones en **Set** sobre argumento principal de tipo **Prop**.
- El argumento principal es solamente usado para asegurar la terminación del algoritmo.
- Inducción bien fundada basada en definición inductiva de accesibilidad:

Inductive Acc (A:Set) (R: A $\rightarrow$ A $\rightarrow$ **Prop**) : A $\rightarrow$ **Prop :=**
    Acc_intro : forall x:A, (forall y:A, R y x $\rightarrow$ Acc R y) $\rightarrow$ Acc R x

- Intuición: elementos accesibles de una relación son aquellos que no pertenecen a una R-cadena infinita decreciente

Accesibilidad y Relaciones bien fundadas

```
Acc_inv:  forall (A : Set) (R : A → A → Prop) (x : A),
              Acc R x → forall y : A, R y x → Acc R y
```

- La prueba `Acc_inv A R x H y Hr` es estructuralmente menor que `H`, por lo tanto puede ser usada como un argumento en una invocación recursiva de una función cuyo argumento principal es `H`

```
Fixpoint Acc_iter (A:Set) (R: A → A → Prop ) (P: A → Set)
    (f: forall x:A, (forall y:A, R y x → P y) → P x)(x:A)(H:Acc R x)
    {struct H} : P x  :=
    f x (fun (y:A)(Hr: R y x) => Acc_iter P f (Acc_inv H y Hr)).
```

Accesibilidad y Relaciones bien fundadas (2)

Definition well_founded : (A : Set) (R : A → A → Prop) : Prop :=
forall a : A, Acc R a

- Si una relación es bien fundada, la misma puede ser usada para definir funciones recursivas: pueden ser definidas de tal forma que la relación es usada para controlar cuales son invocaciones (recursivas) correctas.

Definition well_founded_induction (A:Set) (R: A → A → **Prop**)
(Hwf: well_founded R) (P: A → **Set**)
(f: forall x:A, (forall y:A, R y x → P y) → P x)(x:A) : P x :=
Acc_iter P f (Hwf x).

Pruebas de buena fundación

Ciertos órdenes pueden obtenerse a través de operadores de composición de relaciones que garantizan que si las relaciones iniciales son bien fundadas, el resultado de la composición también lo es [Paulson86].

Ejemplo típico de orden bien fundado: $< \subseteq N \times N$

Operadores de buena fundación

Subrelación :

Si **R** $\subseteq$ AxA es bien fundada y **R´**$\subseteq$ R

*entonces **R´** es bien fundada*

Imagen Inversa:

Sean

– **f** : A $\rightarrow$ B

– **R** $\subseteq$ BxB bien fundada

Definimos

– **R***$\subseteq$ AxA como x **R***y **ssi** f (x) **R** f(y)

*entonces **R*** es bien fundada*

Operadores de buena fundación

Clausura transitiva

Si $\mathbf{R} \subseteq AxA$ es bien fundada, entonces $\mathbf{R}^+ \subseteq AxA$ tal que

$x \, \mathbf{R}^+ y$ **ssi** $(x\mathbf{R}y \ \vee \ \exists z \in A$ tq. $x\mathbf{R}z \ \wedge \ z\mathbf{R}^+y)$

es bien fundada

Suma disjunta

Si $\mathbf{R}_A \subseteq AxA$ es bien fundada y $\mathbf{R}_B \subseteq BxB$ es bien fundada entonces $\mathbf{R}_{A+B} \subseteq (A+B) \times (A+B)$ tal que $a \, \mathbf{R}_{A+B} \, b$ **ssi** $(a,b \in A \ y \ a\mathbf{R}_Ab \ \vee \ a,b \in B \ y \ a\mathbf{R}_Bb)$

es bien fundada

Operadores de buena fundación

Producto Lexicográfico

Si $\mathbf{R_A} \subseteq AxA$ es bien fundada y $\mathbf{R_B} \subseteq BxB$ es bien fundada
entonces $\mathbf{R_{ALexB}} \subseteq (AxB) \times (AxB)$ tal que $(a,b)\ \mathbf{R_{ALexB}}\ (a',b')$
ssi $(\ a\mathbf{R_A}a' \vee\ a=a' \wedge\ b\mathbf{R_B}b'\ )$

es bien fundada

Nota:

el producto lexicográfico se generaliza para el caso en el
que B es una familia de tipos indexada por A (o sea B: A$\rightarrow$Set).
En ese caso el orden $\mathbf{R_{ALexB}} \subseteq \mathbf{\Sigma_{x:A}\ B(x) \times \Sigma_{x:A}\ B(x)}$
Este producto general es el que está definido en Coq.

Operadores de buena fundación en Coq

- En las bibliotecas se encuentran:
  - Ordenes de base bien fundados
    - por ejemplo lt en nat está en **theories\ARITH\Wf_nat**
  - Operadores para construir órdenes bien fundados
    - **theories\WELLFOUNDED** (con la prueba respectiva de que construyen órdenes bien fundados)