

# **6. Verificación de** **programas funcionales**

# Especificación de programas

- $\forall i \in \text{Input}. \psi(i) \rightarrow \exists o \in \text{Output}. \phi(i, o)$

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

**En general:**

- $\forall i_1 \in \text{Input}_1 \dots \forall i_n \in \text{Input}_n. \psi(i_1 \dots i_n) \rightarrow \exists o \in \text{Output}. \phi(i_1, \dots, i_n, o)$

**Ejemplo:**

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

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

# Especificaciones en Coq

## En Coq:

$\text{forall } (i_1:\text{Input}_1) \dots (i_n:\text{Input}_n), \psi(i_1 \dots i_n) \rightarrow \{o:\text{Output} \mid \varphi(i_1, \dots, i_n, o)\}$

## Ejemplos:

–  $\text{forall } (a \ b:\text{nat}), \ b \neq 0 \rightarrow \{qr: \mathbb{N}^* \mathbb{N} \mid \text{match } qr \text{ with}$   
 $(q,r) \Rightarrow a = b.q + r \wedge r < b$   
 $\text{end } \}$

– ¿Especificación de la función que ordena una lista?

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

# Extracción de programas

Si la prueba del existencial

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

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

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

el *contenido computacional* de  $p$  será una función

$$f_p : \text{Input} \rightarrow \text{Output}$$

Para ello deberá ocurrir que :

Input : **Set**

Output : **Set**

$\psi(i), \varphi(i, o) : \text{Prop}$

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**                      **{ x : A | P x }**  
:= exist : forall x:A, P x → sig A P.

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

---

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

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

# 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**

:= left : A → (sumbool A B)

| right : B → (sumbool A B)

**{A } + {B }**

Inductive sumor (A:**Set**)(B: **Prop**) : **Set**

:= left : A → (sumor A B)

| right : B → (sumor A B)

**A + {B}**

Inductive or (A B: **Prop**) : **Prop**

:= or\_introl : A → (or A B)

| or\_intror : B → (or A B)

**A ∨ B**

# Mecanismo de extracción en Coq

- **Extracción del programa**

**Extraction** *id*

**Recursive Extraction**  $id_1 \dots id_n$

**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**  $type_{coq} \Rightarrow type_{leng} [c1_{ml} \dots cn_{leng}]$

**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

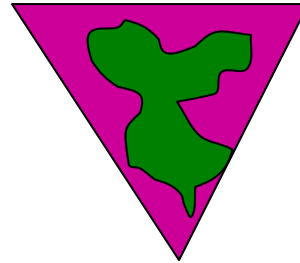
Es el proceso inverso al de extracción

**P: Spec**



**f<sub>P</sub>**

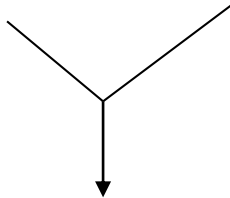
**Programa**



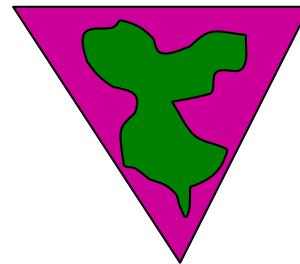
**: Spec**

**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  $b \in \mathbf{N}$ ,  $b \neq 0$ .

Calculamos  $a \mathbf{div} b$  y  $a \mathbf{mod} b$  *simultáneamente* haciendo recursión en  $a$ :

- $0 \mathbf{divmod} b = (0,0)$
- $(n+1) \mathbf{divmod} b = \text{let } (q,r) = n \mathbf{divmod} b$   
                                  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  $a$ :
  - $0 \text{ divmod } b = \langle 0, 0 \rangle \rightarrow 0 = b.0 + 0$  y  $0 < b$
  - $(n+1) \text{ divmod } b =$ 

$\text{let } (q, r) := n \text{ divmod } b$ $\text{in if } r < b - 1$ $\text{then } (q, r+1)$ $\text{else } (q+1, 0)$	$\rightarrow$ $\text{supongamos } n = b.q + r \text{ y } r < b.$ $\text{luego, si } r < b - 1$ $\text{entonces } n+1 = b.q + (r+1) \text{ y } r+1 < b$ $\text{sino } n+1 = b.(q+1) + 0 \text{ y } 0 < b$
--	---

# Algunas Tácticas

- **refine *term***: esta táctica permite proveer una prueba “casi” exacta de un goal (término con “holes”, denotados `_`)
- **functional induction *id term*<sub>1</sub> ... *term*<sub>*n*</sub>**: 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 → A → **Prop**) : A → **Prop** :=  
 Acc\_intro : forall x:A, (forall y:A, R y x → Acc R y) → 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

$\text{Acc\_inv: forall (A : Set) (R : A \rightarrow A \rightarrow \text{Prop}) (x : A),$   
 $\text{Acc R x} \rightarrow \text{forall y : A, R y x} \rightarrow \text{Acc R y}$

- La prueba  $\text{Acc\_inv A R x H y Hr}$  es estructuralmente menor que  $\text{H}$ , por lo tanto puede ser usada como un argumento en una invocación recursiva de una función cuyo argumento principal es  $\text{H}$

$\text{Fixpoint Acc\_iter (A:Set) (R: A \rightarrow A \rightarrow \text{Prop} ) (P: A \rightarrow \text{Set})}$   
 $(f: \text{forall x:A, (forall y:A, R y x} \rightarrow \text{P y}) \rightarrow \text{P x})(x:A)(H:\text{Acc R x})$   
 $\{\text{struct H}\} : \text{P x} :=$   
 $f x (\text{fun (y:A)(Hr: R y x)} \Rightarrow \text{Acc\_iter P f (Acc\_inv H y Hr)}).$

# 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:**  $< \subseteq \mathbb{N} \times \mathbb{N}$

# Operadores de buena fundación

## Subrelación :

Si  $R \subseteq A \times A$  es bien fundada y  $R' \subseteq R$   
entonces  $R'$  es bien fundada

## Imagen Inversa:

Sean

- $f : A \rightarrow B$
- $R \subseteq B \times B$  bien fundada

Definimos

- $R^* \subseteq A \times A$  como  $x R^* y \text{ ssi } f(x) R f(y)$   
entonces  $R^*$  es bien fundada

# Operadores de buena fundación

## Clausura transitiva

Si  $R \subseteq A \times A$  es bien fundada, entonces  $R^+ \subseteq A \times A$  tal que  
 $x R^+ y$  ssi  $(x R y \vee \exists z \in A \text{ tq. } x R z \wedge z R^+ y)$   
*es bien fundada*

## Suma disjunta

Si  $R_A \subseteq A \times A$  es bien fundada y  $R_B \subseteq B \times B$  es bien fundada  
entonces  $R_{A+B} \subseteq (A+B) \times (A+B)$  tal que  $a R_{A+B} b$  ssi  
 $(a, b \in A \text{ y } a R_A b \vee a, b \in B \text{ y } a R_B b)$   
*es bien fundada*

# Operadores de buena fundación

## Producto Lexicográfico

Si  $R_A \subseteq A \times A$  es bien fundada y  $R_B \subseteq B \times B$  es bien fundada entonces  $R_{A \text{Lex} B} \subseteq (A \times B) \times (A \times B)$  tal que  $(a, b) R_{A \text{Lex} B} (a', b')$  ssi  $(a R_A a' \vee a = a' \wedge b R_B b')$   
*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  $B: A \rightarrow \text{Set}$ ). En ese caso el orden  $R_{A \text{Lex} B} \subseteq \Sigma_{x:A} B(x) \times \Sigma_{x:A} B(x)$   
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)