Introducción a los tipos dependientes y a la programación verificada en Agda

Marcelo Lynch

Instituto Tecnológico de Buenos Aires

16 de diciembre de 2019

Programación verificada

- Se está volviendo cada vez más importante garantizar que el software *hace lo que queremos que haga*.
- Los *bugs* pueden ser muy costosos y hasta costar vidas
- **Objetivo**: que nuestros programas y algoritmos sean *correctos* de acuerdo a alguna *especificación*

Técnicas de verificación

Existen varias maneras de aproximarse a este objetivo:

- Testing y revisión de pares
- Análisis estático y dinámico para encontrar bugs
- Verificación formal
  - Model checking
  - Verificación deductiva: demostradores de teoremas

Verificación formal

- Los **métodos formales** nos permiten **demostrar la corrección con rigor matemático**.
- Podemos describir a los métodos formales como "la matemática aplicada para modelar y analizar sistemas de software".
- Esta verificación se hace proveyendo una demostración formal de un modelo matemático del sistema.
  - La correspondencia entre el modelo matemático y el sistema real se presume.

Demostradores de teoremas

El nivel de automaticidad de estas herramientas nos deja categorizarlas en:

- Demostradores automáticos
- Asistentes de demostraciones
- Verificadores de demostraciones

Asistentes de demostraciones

- Incluyen una interfaz en la que el usuario guía de alguna manera la búsqueda de la demostración
- Se habla en estos casos de una "colaboración humano-máquina"
- Ejemplos notables:
  - Coq
  - Isabelle
  - Agda

Agda - Introducción

- Agda es un lenguaje funcional con *tipos dependientes*, y sintaxis similar a Haskell
- Al mismo tiempo es un asistente de demostraciones que funciona dentro del mismo lenguaje (mediante su type checker)
- Desarrollado mayormente por Ulf Norell para su tesis de doctorado en la Universidad Tecnológica Chalmers en Gotemburgo, Suecia

Agda y Haskell

- Agda está desarrollado en Haskell
- La sintaxis de Agda esta inspirada en Haskell y el estilo de programación es similar
- Tienen distintas teorías subyacentes
  - Haskell está basado en System F (lambda cálculo tipado con polimorfismo paramétrico)
  - Agda está basado en la teoría de tipos dependientes de Martin-Löf
- Agda posee un **termination checker**: todos los programas deben terminar. Es decir, no hay recursión general y no es Turing-completo.

Teorías de tipos

- Se llama **teoría de tipos** a una serie de sistemas formales que sirven como alternativa a la teoría de conjuntos como fundamento formal de la matemática.
- Los matemáticos y científicos de la computación trabajan siempre con *construcciones* u *objetos* teóricos.
- Implícitamente se suele categorizar a los objetos con los que trabaja, asociándolos a un *tipo*.
- Las teorías de tipos hacen explícita esta asociación, tratando a los distintos tipos como "ciudadanos de primera clase"

La teoría de tipos de Martin-Löf

- Presentada por Per Martin-Löf como un fundamento matemático intuicionista (constructivista): por esto es también conocida como *teoría de tipos intuicionista*.
- La descripción de la teoría y sus elementos se hace a partir de *juicios*, es decir afirmaciones en el lenguaje metalógico (por afuera del sistema)
  - Así, por ejemplo, en lugar de *definir* el concepto de tipo veremos en qué condiciones se puede decir (es decir, emitir un juicio, o "explicar") que algo *es un tipo*.

Tipos

Podemos emitir el juicio "A **es un tipo**" cuando conocemos:

- Cuándo un objeto pertence al tipo (las condiciones de pertenencia)
- que significa que dos objetos del tipo sean iguales (mediante una relación de equivalencia)

Tipos

- La pertenencia de un objeto a al tipo A se nota $a \in A$ o $a : A$.
- Decimos que B es una *familia de tipos* indexada por A si B es una asignación que para cada $a \in A$ asigna un tipo $B(a)$.

Términos y objetos

- La teoría tiene asociada una noción de cómputo, que es relevante al contextualizarla dentro de un lenguaje de programación
- En rigor en lugar de objetos hablamos de **términos**
  - Por ejemplo, 4, $2 + 2$ y $2 \cdot 2$ podrían ser términos distintos del tipo **Nat**

Términos y objetos

- La teoría tiene asociada una noción de cómputo, que es relevante al contextualizarla dentro de un lenguaje de programación
- En rigor en lugar de objetos hablamos de **términos**
  - Por ejemplo, 4, $2 + 2$ y $2 \cdot 2$ podrían ser términos distintos del tipo **Nat**
- Existen **reglas de reescritura** sobre los términos:
  - Por ejemplo, $2 + 2$ se puede reescribir mediante una serie de reglas de reescritura al término 4
  - Decimos que $2 + 2$ *reduce a* 4
  - Podemos pensar a dos términos que se pueden reducir a un mismo término como "iguales" en cierto sentido: esta noción se llama *igualdad definicional*

Ejemplos de tipos: **Set**

- La colección de todos los conjuntos forma un tipo, que podemos llamar **Set**.
- También podemos pensar al tipo **Set** como un tipo que contiene a otros tipos (pensando a los tipos estructuralmente como conjuntos de objetos). Esta es la noción que usamos en Agda, donde si A es un tipo entonces $A : Set$.

Ejemplos de tipos: **Set**

- La colección de todos los conjuntos forma un tipo, que podemos llamar **Set**.
- También podemos pensar al tipo **Set** como un tipo que contiene a otros tipos (pensando a los tipos estructuralmente como conjuntos de objetos). Esta es la noción que usamos en Agda, donde si A es un tipo entonces $A : Set$.
  - Observación: Set es un tipo, pero el juicio $Set : Set$, lleva a la clásica inconsistencia de Bertrand Russell.
  - Esto se puede arreglar introduciendo una jerarquía de tipos Set_0, Set_1, ..., con **Set** $= Set_0$ y $Set_i : Set_{i+1}$. No nos adentraremos demasiado en esto, y manejamos a continuación tipos de "primer nivel" que estan en **Set**.

Tipos funcionales

- Si A y B son tipos podemos introducir el tipo $A \to B$, el espacio de funciones de A a B.
- Un elemento $f \in A \to B$ se puede *aplicar* a cualquier $a \in A$, y tenemos $f(b) \in B$.

Pares ordenados

- Si A y B son tipos podemos introducir el tipo $A \times B$ de pares ordenados: la primera componente tiene elementos de A y la segunda elementos de B.
- Notamos a un elemento de $A \times B$ como (a, b).
- Los tipos de pares ordenados vienen equipados con proyecciones π_1, π_2 funciones tales que $\pi_1((a, b)) = a$ y $\pi_2((a, b)) = b$.

El tipo suma

- Si A y B son tipos podemos introducir el tipo $A + B$, la suma disjunta de A y B.
- Un elemento de este tipo será o bien un elemento de A o uno de B, junto con una indicación de si provino de A o de B
- Todos los elementos de $A + B$ pueden describirse mediante dos funciones $inj_A : A \to A + B$ e $inj_B : B \to A + B$.
- En Haskell el concepto análogo es el the la forma del tipo `Either a b = Left a | Right b`, donde podemos saber el "origen" por pattern matching en los constructores `Left` y `Right` (estos constructores son análogos a las funciones *inj*).

Tipos dependientes

- El concepto central de esta teoría de tipos es el de **tipo dependiente**
- La definición de un tipo dependiente *depende de un valor* (y no de otros tipos como lo que veníamos viendo).
- Veremos que la existencia de tipos dependientes es lo que nos deja hacer demostraciones usando esta teoría de tipos

Funciones dependientes

Si B es una familia de tipos sobre A, existe el llamado producto dependiente, tipo de funciones dependientes o *tipo pi*:

$$\prod_{a \in A} B(a)$$

Este tipo contiene funciones con dominio (o entrada) en A pero cuyo codominio (o tipo de salida) depende del valor en la que se aplica la función.

Funciones dependientes (tipos Π)

Por ejemplo: si llamamos $VecN(n)$ al conjunto de listas de n elementos naturales, podemos considerar una función f que aplicada en un numero natural n resulta en una lista de n ceros. Con esto:

$$f \in \prod_{n \in \mathbb{N}} VecN(n)$$

Notemos que cuando B es una asignación constante, este tipo corresponde al tipo de funciones $A \to B$ antes mencionado (donde el codominio no depende del valor de entrada).

Pares dependientes (tipos Σ)

Los elementos de los tipos Σ son pares ordenados donde el tipo de la segunda componente depende del valor en la primera componente.
Esto es, si B es una familia de tipos sobre A, existe el tipo de pares dependientes:

$$\sum_{a \in A} B(a)$$

Cuando B es una asignación constante, este tipo corresponde al tipo de pares ordenados $A \times B$.

El tipo igualdad

- Dados dos términos x, y puede introducirse el tipo igualdad $x \equiv y$.
- Existe un único constructor `refl` para cada tipo A que dado un objeto de A devuelve un valor de $a \equiv a$:

$$\texttt{refl} \in \prod_{a \in A} a \equiv a$$

El tipo igualdad

- Notemos que `refl` es la única forma de construir (encontrar, describir) un valor del tipo igualdad
- Esto significa que si bien podemos hablar de un tipo $x \equiv y$ para cualesquiera dos términos x, y, el tipo $x \equiv y$ solo estará habitado si x es igual a y.

¿El tipo igualdad?

Con la introducción del tipo igualdad nos acercamos a la idea central que nos permite demostrar cosas en lenguajes como Agda: estamos diciendo que existe un **tipo** que representa a la **igualdad**, que es una afirmación lógica.

Veremos ahora precisamente como se establece este paralelismo entre tipos y fórmulas lógicas

Lógica intuicionista

- En la lógica clásica vale el principio del tercero excluido: "o bien A es verdadero, o bien A es falso".
- La lógica intuicionista rechaza el principio del tercero excluido: exige una demostración concreta o bien de que A es verdadero o de que B es verdadero para concluir que $A \vee B$ es verdadero: en otras palabras, con la demostración de $A \vee B$ debemos saber *cuál de los dos vale*.
- La lógica intuicionista es *constructivista*: para demostrar la existencia de un elemento que satisface una propiedad hay que construirlo, exhibirlo.

Proposiciones como tipos

El concepto de *proposiciones como tipos*, o *correspondencia de Curry-Howard* es la relación directa que existe entre las fórmulas de la lógica (proposiciones) con los tipos en la teoría de tipos.

Lógica	**Teoría de tipos**
Proposición, fórmula	Tipo
Demostración	Programa
Evidencia	Habitante de un tipo

Cuadro: Correspondencia de Curry-Howard

Proposiciones como tipos

Aquí ya vemos como la correspondencia se da en el marco de la lógica intuicionista:

- La *evidencia* la da un habitante concreto de un tipo
- Una demostración se da con un *programa*, que es una **verdadera construcción de la evidencia**

La correspondencia de Curry-Howard

- $A \wedge B$ corresponde a $A \times B$
- $A \vee B$ corresponde a $A + B$
- $A \Rightarrow B$ corresponde a $A \to B$
- $\forall x : A.\ B(x)$ corresponde a $\prod_{x \in A} B(x)$
- $\exists x : A.\ B(x)$ corresponde a $\sum_{x \in A} B(x)$

Demostrando con el sistema de tipos

Podemos imaginar ahora cómo se demuestran las propiedades de los programas en un lenguaje con tipos dependientes como Agda: **simplemente debemos exhibir un elemento que habite el tipo que corresponde a la proposición que queremos demostrar**.

Es decir, para demostrar una propiedad como

$$\forall n : \mathbb{N} \,.\, \forall m : \mathbb{N} \,.\, n + m = m + n$$

basta con exhibir (construir) un elemento del tipo

$$\prod_{n \in \mathbb{N}} \prod_{m \in \mathbb{N}} n + m \equiv m + n$$

Programando en Agda - Definiciones

```
-- Definimos los booleanos
data 𝔹 : Set where
  tt : 𝔹 -- true
  ff : 𝔹 -- false

-- Un tipo definido por inducción: los naturales
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

-- Un tipo polimórfico: lista de A
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A

-- Pragmas
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN LIST List #-}
```

Funciones

```
-- And booleano
_&&_ : 𝔹 → 𝔹 → 𝔹
tt && b = b
ff && _ = ff

-- Suma de naturales
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)

if_then_else_ : {A : Set} → 𝔹 → A → A → A
if tt then a else _ = a
if ff then _ else a = a

pred : ℕ → ℕ
pred 0 = 0
pred (suc n) = n
```

Tipos dependientes

```
data Vec012 (A : Set) : (n : ℕ) → Set where
  v012[] : Vec012 A 0
  v012[_] : A → Vec012 A 1
  v012[_,_] : A → A → Vec012 A 2
```

Tipos dependientes

```
data 𝕍 (A : Set) : ℕ → Set where
  [] : 𝕍 A 0
  _::_ : ∀ {n : ℕ} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)
```

La igualdad

```
data Eq (A : Set) (x : A) : A → Set where
  refl : (Eq A x) x
```

La igualdad

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x


-- Asociamos ≡ a la noción interna de igualdad de Agda
{-# BUILTIN EQUALITY _≡_ #-}
infix 3 _≡_
```

Igualdad proposicional y definicional

```
1+1-es-2 : 1 + 1 ≡ 2
1+1-es-2 = refl

pred3 : pred 3 ≡ 2
pred3 = refl

pred3' : pred 3 ≡ pred (suc 2)
pred3' = refl
```

Algunas propiedades de la igualdad

```
-- La igualdad es simétrica
sym : ∀ {A : Set}{x y : A}
        → x ≡ y → y ≡ x
sym refl = refl
```

```
-- La igualdad es una congruencia
cong : ∀ {A B : Set} (f : A → B) {x y}
         → x ≡ y → f x ≡ f y
cong f refl = refl
```

Demostraciones

```
-- Negación
~_ : 𝔹 → 𝔹
~ tt = ff
~ ff = tt

-- Demostrando un enunciado universalmente cuantificado
-- En este caso basta con enumerar
-- todos los casos por pattern matching
~~-elim : ∀ (b : 𝔹) → ~ ~ b ≡ b
~~-elim tt = refl
~~-elim ff = refl

-- Reutilizando demostraciones con rewrite
~~&& : ∀ (b1 b2 : 𝔹) → (~ ~ b1) && b2 ≡ b1 && b2
~~&& b1 b2 rewrite (~~-elim b1) = refl
```

Demostraciones

```
_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys       = ys
(x :: xs) ++ ys = x :: (xs ++ ys)


-- Con rewrite
++-r-identity : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-r-identity [] = refl
++-r-identity (x :: xs) rewrite ++-r-identity xs = refl
```

Demostraciones

```
_++_ : ∀ {A : Set} → List A → List A → List A
[] ++ ys       = ys
(x :: xs) ++ ys = x :: (xs ++ ys)


-- Con rewrite
++-r-identity : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-r-identity [] = refl
++-r-identity (x :: xs) rewrite ++-r-identity xs = refl


-- Usando razonamientos
++-r-identity' : ∀ {A : Set} (xs : List A) → xs ++ [] ≡ xs
++-r-identity' [] = refl
++-r-identity' (x :: xs) =
                begin
                   (x :: xs) ++ []
                ≡⟨⟩
                  x :: (xs ++ [])
                ≡⟨ cong (x ::_) (++-r-identity xs) ⟩
                 x :: xs
                ∎
```

Razonamientos con igualdad

```
-- Cosmético
begin_ : ∀ {x y : A}
  → x ≡ y
    -----
  → x ≡ y
begin x≡y  =  x≡y
```

```
-- Me deja encadenar cosas definicionalmente iguales
_≡⟨⟩_ : ∀ (x : A) {y : A}
  → x ≡ y
    -----
  → x ≡ y
x ≡⟨⟩ x≡y  =  x≡y
```

Razonamientos con igualdad

```
-- El argumento del medio es 'justificación' (mediante transitividad)
-- para decir que el de la izquierda y la derecha son iguales
  _≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
      -----
    → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z  =  trans x≡y y≡z


-- Cosmético
  _∎ : ∀ (x : A)
      -----
    → x ≡ x
  x ∎  =  refl
```

Más demostraciones

```
-- Asociatividad de append
++-assoc : ∀ {A : Set} (xs ys zs : List A)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)

++-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  ≡⟨⟩
    ys ++ zs
  ≡⟨⟩
   [] ++ (ys ++ zs)
  ∎

++-assoc (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  ≡⟨⟩                                        -- Definición de ++
    (x :: (xs ++ ys)) ++ zs
  ≡⟨⟩                                        -- Definición de ++
    x :: ((xs ++ ys) ++ zs)
  ≡⟨ cong (x ::_) (++-assoc xs ys zs) ⟩      -- Hipótesis inductiva
    x :: (xs ++ (ys ++ zs))
  ≡⟨⟩                                        -- Definición de ++
    (x :: xs) ++ (ys ++ zs)
  ∎
```

Más demostraciones

```
-- Reverse lento: O(N²)
reverse : ∀ {A : Set} → List A → List A
reverse []       =  []
reverse (x :: xs)  =  reverse xs ++ [ x ]

-- Reverse rápido (helper)
fastrev : ∀ {A : Set} → List A → List A → List A
fastrev [] ys  =  ys
fastrev (x :: xs) ys  =  fastrev xs (x :: ys)

-- Reverse  rápido: O(N)
rev : ∀ {A : Set} → List A → List A
rev xs = fastrev xs []
```

Quiero demostrar que `reverse` y `rev` hacen lo mismo...

Más demostraciones

```
-- Demostremos este lema
fastrev-lem : ∀ {A : Set} (xs ys : List A)
                → fastrev xs ys ≡ reverse xs ++ ys

-- Caso base
fastrev-lem [] ys =
  begin
    fastrev [] ys
  ≡⟨                                    -- Definicion de fastrev
    ys
  ≡⟨                                    -- Definición de reverse
    reverse [] ++ ys
  ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys

      ≡⟨⟩
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩
        fastrev xs (x :: ys)
```

-- Definicion de fastrev

```
      ≡⟨⟩
        reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩
        fastrev xs (x :: ys)                          -- Definicion de fastrev
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                   -- Hipótesis inductiva
        reverse xs ++ (x :: ys)

      ≡⟨⟩
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩                                          -- Definicion de fastrev
        fastrev xs (x :: ys)
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                -- Hipótesis inductiva
        reverse xs ++ (x :: ys)

      ≡⟨⟩
       (reverse xs ++ [ x ]) ++ ys
      ≡⟨⟩                                          -- Definición de reverse
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩                                          -- Definicion de fastrev
        fastrev xs (x :: ys)
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                -- Hipótesis inductiva
        reverse xs ++ (x :: ys)


      ≡⟨⟩
        reverse xs ++ ([ x ] ++ ys)
      ≡⟨ sym (++-assoc (reverse xs) ([ x ]) ys) ⟩  -- Asociatividad de ++
        (reverse xs ++ [ x ]) ++ ys
      ≡⟨⟩                                          -- Definición de reverse
        reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩                                          -- Definicion de fastrev
        fastrev xs (x :: ys)
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                -- Hipótesis inductiva
        reverse xs ++ (x :: ys)


      ≡⟨⟩
       reverse xs ++ ((x :: []) ++ ys)
      ≡⟨⟩                                          -- Notación: x :: [] ≡ [ x ]
       reverse xs ++ ([ x ] ++ ys)
      ≡⟨ sym (++-assoc (reverse xs) ([ x ]) ys) ⟩  -- Asociatividad de ++
       (reverse xs ++ [ x ]) ++ ys
      ≡⟨⟩                                          -- Definición de reverse
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩                                          -- Definicion de fastrev
        fastrev xs (x :: ys)
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                -- Hipótesis inductiva
        reverse xs ++ (x :: ys)

       reverse xs ++ (x :: ([] ++ ys))
      ≡⟨⟩                                          -- Definición de ++ (2)
       reverse xs ++ ((x :: []) ++ ys)
      ≡⟨⟩                                          -- Notación: x :: [] ≡ [ x ]
       reverse xs ++ ([ x ] ++ ys)
      ≡⟨ sym (++-assoc (reverse xs) ([ x ]) ys) ⟩  -- Asociatividad de ++
       (reverse xs ++ [ x ]) ++ ys
      ≡⟨⟩                                          -- Definición de reverse
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Paso inductivo
fastrev-lem (x :: xs) ys =
      begin
        fastrev (x :: xs) ys
      ≡⟨⟩                                                    -- Definicion de fastrev
        fastrev xs (x :: ys)
      ≡⟨ fastrev-lem xs (x :: ys) ⟩                           -- Hipótesis inductiva
        reverse xs ++ (x :: ys)
      ≡⟨⟩                                                    -- Definicion de ++ (1)
       reverse xs ++ (x :: ([] ++ ys))
      ≡⟨⟩                                                    -- Definición de ++ (2)
       reverse xs ++ ((x :: []) ++ ys)
      ≡⟨⟩                                                    -- Notación: x :: [] ≡ [ x ]
       reverse xs ++ ([ x ] ++ ys)
      ≡⟨ sym (++-assoc (reverse xs) ([ x ]) ys) ⟩   -- Asociatividad de ++
       (reverse xs ++ [ x ]) ++ ys
      ≡⟨⟩                                                    -- Definición de reverse
       reverse (x :: xs) ++ ys
      ∎
```

Más demostraciones

```
-- Demostración: reverse y rev hacen lo mismo para cualquier argumento
reverses-equal-app : ∀ {A : Set} (xs : List A) → rev xs ≡ reverse xs
reverses-equal-app xs =
  begin
    rev xs
  ≡⟨⟩
    fastrev xs []
  ≡⟨ fastrev-lem xs [] ⟩                    -- Aplico el lema
    reverse xs ++ []
  ≡⟨ ++-r-identity (reverse xs) ⟩           -- xs + [] ≡ xs
    reverse xs
  ∎
```

Extensionalidad como postulado

```
postulate
  extensionality : ∀ {ℓ} {A B : Set ℓ} {f g : A → B}
    → (∀ (x : A) → f x ≡ g x)
    → f ≡ g
```

- No se puede demostrar extensionalidad dentro de Agda
- Sin embargo el postulado es consistente con Agda así que podemos usarlo sin problemas

Extensionalidad como postulado

```
postulate
  extensionality : ∀ {ℓ} {A B : Set ℓ} {f g : A → B}
    → (∀ (x : A) → f x ≡ g x)
    → f ≡ g
```

```
-- Con extensionalidad puedo probar la igualdad de las funciones
reverses-equal : ∀ {A : Set} → rev {A} ≡ reverse {A}
reverses-equal = extensionality reverses-equal-app
```

Verificación interna

- Hasta ahora definimos tipos de datos (como `List`) y programas (como $++$) y luego demostramos propiedades sobre los mismos.
  - Esto podría llamarse *verificación externa*: las demostraciones son externas a los programas.
- En contraste, podemos considerar un estilo de verificación que podemos llamar *verificación interna* en donde expresamos las proposiciones *dentro de los mismos tipos de datos y programas*
- La idea es escribir tipos y funciones más expresivos: la corrección de las funciones y las invariantes de las estructuras de datos están garantizadas por el propio tipo.

Vectores: el tipo $\mathbb{V}$

```
data 𝕍 (A : Set) : ℕ → Set where
  [] : 𝕍 A 0
  _::_ : ∀ {n : ℕ} (x : A) (xs : 𝕍 A n) → 𝕍 A (suc n)

-- Concatenación de vectores
_++𝕍_ : ∀ {A : Set} {n m : ℕ} → 𝕍 A n → 𝕍 A m → 𝕍 A (n + m)
[] ++𝕍 ys = ys
(x :: xs) ++𝕍 ys = x :: (xs ++𝕍 ys)

-- Extraer el primer elemento de vectores no vacíos
head𝕍 : ∀ {A : Set} {n : ℕ} → 𝕍 A (suc n) → A
head𝕍 (x :: xs) = x
```

Árboles binarios de búsqueda

$$\_\leq\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$$
$$\mathsf{zero} \leq \mathsf{zero} = \mathsf{tt}$$
$$\mathsf{zero} \leq (\mathsf{suc}\ a) = \mathsf{tt}$$
$$(\mathsf{suc}\ b) \leq \mathsf{zero} = \mathsf{ff}$$
$$(\mathsf{suc}\ a) \leq (\mathsf{suc}\ b) = a \leq b$$

```
-- Árboles binarios de búsqueda con elementos naturales.
-- Indizados por dos elementos naturales, la cota inferior
-- y la superior de los elementos del árbol
data bstℕ : ℕ → ℕ → Set where
  leaf : ∀ {l u : ℕ} → l ≤ u ≡ tt → bstℕ l u
  node : ∀ {ll lr ul ur : ℕ}
           (elem : ℕ) → bstℕ ll ul → bstℕ lr ur →
           ul ≤ elem ≡ tt → elem ≤ lr ≡ tt →
           bstℕ ll ur
```

Conclusiones

- Conclusiones respecto de Agda
  - Interesante e importante conocer el costado teórico
  - Experiencia interactiva muy rica
  - Intuitivo, estilo parecido a demostración en papel
- Otras aproximaciones a la programación verificada
  - Coq
  - Haskell
  - Idris