

Introducción a la Verificación Formal

Clase 3 – 11/09/2025

- Repaso Clase 2

Funciones recursivas

- Al definir `let rec f (x1:t1) : t = ... f ...`
 - Dentro de la definición, el cuerpo se chequea *asumiendo* que `f` tiene tipo $(x1:t1) \rightarrow t$
(como es usual)

Pero, tenemos que chequear terminación, con lo cual hay un pequeño ajuste:

$$(x1':t1\{x1' \ll x1\}) \rightarrow t$$

es decir: la función recursivamente ligada sólo está definida para `x1'` “menores” al `x1` original

- La relación `<<` es un orden bien fundado (sin cadenas infinitamente descendientes) sobre todos los valores de F^* . En particular:
 - Para `x, y` naturales $x \ll y \iff x < y$
 - Dado un constructor `C`, tenemos $x_i \ll (C \ x_1 \ \dots \ x_i \ \dots \ x_n)$

Claúsula decreases

- Siempre debe haber una métrica de terminación.
 - Si no se explicita, F* toma el orden lexicográfico de todos los argumentos

```
let rec add (m n : nat) : nat =  
  if n = 0 then m  
  else add (m+1) (n-1)
```

- Para darla explícitamente se usa la cláusula decreases:

```
let rec add (m n : nat) : Tot nat (decreases n) =  
  if n = 0 then m  
  else add (m+1) (n-1)
```

Orden lexicográfico

- Orden sobre tuplas a partir de órdenes para los componentes

$$(x_1, y_1, z_1) < (x_2, y_2, z_2) \iff x_1 < x_2 \vee$$

$$(x_1 == x_2 \wedge y_1 < y_2) \vee$$

$$(x_1 == x_2 \wedge y_1 == y_2 \wedge z_1 < z_2)$$

- La función de Ackermann termina porque respeta el orden lexicográfico en (m, n)

$$A(0, n) = n + 1$$

$$A(m + 1, 0) = A(m, 1)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

Tipos inductivos

- Tipos de datos generados por constructores, que pueden mencionar recursivamente al tipo.
 - Soportan pattern matching, como es usual
- En la definición de listas, la variable a es un *parámetro* del tipo inductivo
 - Es una elección “externa” que no cambia.
 - Hay distintas “versiones” del tipo lista para cada a

```
type list (a:Type) =  
  | Nil : list a  
  | Cons : hd:a -> tl:list a -> list a
```

```
Nil?      : #a:Type -> xs:list a -> bool  
Cons?     : #a:Type -> xs:list a -> bool  
Cons?.hd  : #a:Type -> xs:list a{Cons? xs} -> a  
Cons?.tl  : #a:Type -> xs:list a{Cons? xs} -> list a
```

Tipos indexados

- Además, un tipo puede tener *índices*, que pueden variar de constructor en constructor
 - El pattern matching es consciente de los índices (el match de la imagen es completo)
 - Es la idea principal de GADTs en Haskell

```
type t : bool -> Type =
  | A : nat -> t true
  | B : string -> t false

let f (x : t true) : nat =
  match x with
  | A n -> n
```

GADT para un mini lenguaje

```
type l_ty =  
  | Int  
  | Bool  
  
type expr : l_ty -> Type =  
  | EInt : int -> expr Int  
  | EBool : bool -> expr Bool  
  | EAdd : expr Int -> expr Int -> expr Int  
  | EEq : expr Int -> expr Int -> expr Bool  
  | EIf :  
    #ty: _ ->  
    expr Bool -> expr ty -> expr ty -> expr ty
```

```
EEq : expr Int -> expr Int -> expr Bool  
EI  
#t  
ex  
- Expected type "expr Bool"; but "EInt 0" has type "expr Int"  
Clase3.EInt:  
_0: int -> expr Int  
View Problem (Alt+F8) No quick fixes available  
EIf (EInt 0) (EInt 1) (EInt 2)
```

```
let lift (ty : l_ty) : Type =  
  match ty with  
  | Int -> int  
  | Bool -> bool  
  
val eval (#ty:l_ty) (e : expr ty) : Tot (lift ty)
```


Vectores indexados por longitud

- El tipo `vec` está indexado por un natural que indica su longitud
 - Esto permite implementar `head/tail` de forma segura
 - Idem `append`, etc, que preservan la longitud en el tipo.

```
type vec (a:Type) : nat -> Type =  
  | VNil : vec a 0  
  | VCons : #n:nat -> hd:a -> tl:vec a n -> vec a (n+1)  
  
let vhd (#a:Type) (#n:pos) (xs : vec a n) : a =  
  match xs with  
  | VCons hd tl -> hd
```

Positividad

- No podemos aceptar cualquier definición de tipo inductivo

```
noeq
type ni =
  | Mk : f:(ni -> bool) -> ni

let self_app (x : ni) : bool =
  match x with
  | Mk f -> f x

let boxed : ni =
  Mk self_app

let raro () : bool =
  self_app boxed
```

```
raro ()
~> self_app boxed
~> self_app (Mk self_app)
~> match Mk self_app with
    | Mk f -> f (Mk self_app)
~> self_app (Mk self_app)
~> ....
```

Tareas

- Completar Clase03.*.fst
- Leer capítulos 7, 12, 13
- Otras fuentes:
 - [Conor McBride - Faking it Simulating dependent types in Haskell](#)