Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Deep Embedding de Lenguajes Tipados

Lenguajes Tipados - Deep Embedding

Volviendo al lenguaje con distintos tipos, pero ahora como deep embedding

```
data Expr :: *where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

Evaluador con type-checking

Evaluador parcial, o con type-checking dinámico

```
data Res = RI Int | RB Bool

eval :: Expr → Res
eval (Val n)      = RI n
eval (Add e1 e2) = case (eval e1, eval e2) of
                     (RI n1, RI n2) → RI (n1 + n2)
                     _              → error "type error: Add"
eval (IsZero e)   = case eval e of
                     RI n  → RB (n ≡ 0)
                     _     → error "type error: IsZero"
eval (If e1 e2 e3) = case eval e1 of
                     RB b →
                       case (eval e2, eval e3) of
                         (RI n2, RI n3)   → RI  $ if b then n2 else n3
                         (RB b2, RB b3) → RB $ if b then b2 else b3
                         _              → error "type error: If branches"
                     _     → error "type error: If condition"
```

Type-checking

Podría desacoplar el type-checking de la evaluación

```
data EType = TInt | TBool

infix 6 .::

(.::) :: Expr → EType → Bool
(Val n)      .:: TInt  = True
(Add e1 e2) .:: TInt  = e1 .:: TInt  ∧ e2 .:: TInt
(IsZero e)   .:: TBool = e  .:: TInt
(If c e1 e2) .:: t     = c  .:: TBool ∧ e1 .:: t ∧ e2 .:: t
_            .:: _     = False

wellTyped :: Expr → Bool
wellTyped e = e .:: TInt ∨ e .:: TBool
```

Type-checking

Podría desacoplar el type-checking de la evaluación

```
data EType = TInt | TBool

infix 6 .::

(.::) :: Expr → EType → Bool
(Val n)      .:: TInt  = True
(Add e1 e2) .:: TInt  = e1 .:: TInt  ∧ e2 .:: TInt
(IsZero e)   .:: TBool = e  .:: TInt
(If c e1 e2) .:: t     = c  .:: TBool ∧ e1 .:: t ∧ e2 .:: t
_            .:: _     = False

wellTyped :: Expr → Bool
wellTyped e = e .:: TInt ∨ e .:: TBool
```

y sólo evaluar en caso de tipar correctamente

```
safeEval :: Expr → Res
safeEval e | wellTyped e = eval e
           | otherwise   = error "type error"
```

Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

```
class ExprT (e :: * -> *) where
  valT    :: Int -> e Int
  addT    :: e Int -> e Int -> e Int
  isZeroT :: e Int -> e Bool
  ifT     :: e Bool -> e t -> e t -> e t
```

Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

```
class ExprT (e :: * -> *) where
  valT    :: Int -> e Int
  addT    :: e Int -> e Int -> e Int
  isZeroT :: e Int -> e Bool
  ifT     :: e Bool -> e t -> e t -> e t
```

Necesito empaquetar *Expr* en un $* \to *$ con un phantom type

```
newtype WExpr a = E Expr

instance ExprT WExpr where
  valT n                  = E (Val n)
  addT (E e1) (E e2)      = E (Add e1 e2)
  isZeroT (E e)           = E (IsZero e)
  ifT (E c) (E e1) (E e2) = E (If c e1 e2)
```

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

```
ppExpr (E (Val n))      = show n
ppExpr (E (Add e1 e2)) = "(" ++ ppExpr (E e1) ++ " + " ++ ppExpr (E e2) ++ ")"
ppExpr (E (IsZero e))   = "isZero(" ++ ppExpr (E e) ++ ")"
ppExpr (E (If c e1 e2)) = "if " ++ ppExpr (E c) ++
                          " then " ++ ppExpr (E e1) ++
                          " else " ++ ppExpr (E e2)
```

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

$$
\begin{aligned}
&ppExpr\ (E\ (Val\ n)) &&= show\ n \\
&ppExpr\ (E\ (Add\ e1\ e2)) &&= \texttt{"("} +\!\!+ ppExpr\ (E\ e1) +\!\!+ \texttt{" + "} +\!\!+ ppExpr\ (E\ e2) +\!\!+ \texttt{")"} \\
&ppExpr\ (E\ (IsZero\ e)) &&= \texttt{"isZero("} +\!\!+ ppExpr\ (E\ e) +\!\!+ \texttt{")"} \\
&ppExpr\ (E\ (If\ c\ e1\ e2)) &&= \texttt{"if "} +\!\!+ ppExpr\ (E\ c) +\!\!+ \\
& && \quad \texttt{" then "} +\!\!+ ppExpr\ (E\ e1) +\!\!+ \\
& && \quad \texttt{" else "} +\!\!+ ppExpr\ (E\ e2)
\end{aligned}
$$

Pero no puedo definir algo como

$$
\begin{aligned}
&evalExpr\ (E\ (Val\ n)) &&= n \\
&evalExpr\ (E\ (IsZero\ e)) &&= evalExpr\ (E\ e) \equiv 0 \\
&evalExpr\ (E\ (Add\ e1\ e2)) &&= evalExpr\ (E\ e1) + evalExpr\ (E\ e2) \\
&evalExpr\ (E\ (If\ c\ e1\ e2)) &&= \texttt{if}\ evalExpr\ (E\ c)\ \texttt{then}\ evalExpr\ (E\ e1) \\
& && \quad \texttt{else}\ evalExpr\ (E\ e2)
\end{aligned}
$$

Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

```
ppExpr (E (Val n))      = show n
ppExpr (E (Add e1 e2)) = "(" ++ ppExpr (E e1) ++ " + " ++ ppExpr (E e2) ++ ")"
ppExpr (E (IsZero e))   = "isZero(" ++ ppExpr (E e) ++ ")"
ppExpr (E (If c e1 e2)) = "if " ++ ppExpr (E c) ++
                          " then " ++ ppExpr (E e1) ++
                          " else " ++ ppExpr (E e2)
```

Pero no puedo definir algo como

```
evalExpr (E (Val n))      = n
evalExpr (E (IsZero e))   = evalExpr (E e) == 0
evalExpr (E (Add e1 e2)) = evalExpr (E e1) + evalExpr (E e2)
evalExpr (E (If c e1 e2)) = if evalExpr (E c) then evalExpr (E e1)
                                              else evalExpr (E e2)
```

```
Couldn't match expected type 'Int' with actual type 'Bool'
In the expression: evalExpr (E e) == 0
...
```

Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{aligned}
&\mathit{evalExpr} :: \mathit{WExpr}\ t \to t \\
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n \\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0 \\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1) \\
&&&\quad\ \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{aligned}
&\mathit{evalExpr} :: \mathit{WExpr}\ t \to t \\
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n \\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0 \\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1) \\
& && \quad\ \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

```
Couldn't match expected type 't' with actual type 'Int'
't' is a rigid type variable bound by
  the type signature for:
    evalExpr :: forall t. WExpr t -> t
...
In the expression: n
  In an equation for 'evalExpr': evalExpr (E (Val n)) = n
...
```

Intérpretes Tipados (3)

Se puede definir un *workaround* usando type-clases

```
class Eval t where
  weval :: WExpr t → t

instance Eval Int where
  weval (E (Val n))      = n
  weval (E (Add e1 e2)) = weval (E e1) + weval (E e2)
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)

instance Eval Bool where
  weval (E (IsZero e))   = (weval (E e) :: Int) ≡ 0
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)
```

Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Tipos de Datos Algebraicos Generalizados (GADTs)

Tipo de Datos Algebraico

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

o alternativamente,

```
data Tree :: * → * where
  Leaf  :: Tree a
  Node :: Tree a → a → Tree a → Tree a
```

introduce:

- un nuevo tipo de datos *Tree* de kind $* \rightarrow *$
- Constructores *Leaf* y *Node*
- la posibilidad de usar los constructores en patterns

Restricciones

Los constructores de un tipo de datos T deben:

- resultar en el tipo T
- resultar en un tipo simple
  - $T\ a_1 \ldots a_n$ con $a_1, \ldots, a_n$ variables de tipo distintas

Restricciones

Los constructores de un tipo de datos T deben:

- resultar en el tipo T
- resultar en un tipo simple
  - $T\ a_1 \ldots a_n$ con $a_1, \ldots, a_n$ variables de tipo distintas

Vamos a levantar alguna de estas restricciones.

Deep embedding untyped

```
data Expr :: * where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

Por ejemplo, podemos escribir:

$$If\ (IsZero\ (Add\ (Int\ 0)\ (Int\ 1)))\ (Val\ 3)\ (Val\ 4)$$

que representa la sintaxis abstracta de la siguiente expresión escrita en una sintaxis concreta:

$$\texttt{if}\ isZero\ (0+1)\ \texttt{then}\ 3\ \texttt{else}\ 4$$

Pero podemos escribir también térmimos mal tipados de acuerdo al sistema de tipos del EDSL.

Deep embedding tipado

La idea es codificar el tipo del término que se representa en el propio tipo Haskell.

```
data Expr :: * where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

Deep embedding tipado

La idea es codificar el tipo del término que se representa en el propio tipo Haskell.

```
data Expr :: * where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

```
data Expr :: * → * where
  Val    :: Int → Expr Int
  Add    :: Expr Int → Expr Int → Expr Int
  IsZero :: Expr Int → Expr Bool
  If     :: Expr Bool → Expr t → Expr t → Expr t
```

GADTs

Los GADTs levantan la restricción de que los constructores deben resultar en un tipo simple.

- Los constructores pueden resultar en un subconjunto del tipo
- Consecuencias interesantes en el pattern matching
  - cuando se analiza un *Expr Int*, éste no puede ser construido por *IsZero*
  - cuando se analiza un *Expr Bool*, éste no puede ser construido por *Val* o *Add*
  - cuando se analiza un *Expr Bool*, si encontramos *IsZero* en el pattern, sabemos que tenemos un *Expr Bool*
  - etc

Evaluación usando GADTs: evaluador tagless

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ t \to t \\
&\mathit{eval}\ (\mathit{Val}\ n) &&= n \\
&\mathit{eval}\ (\mathit{Add}\ e1\ e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
&\mathit{eval}\ (\mathit{IsZero}\ e) &&= \mathit{eval}\ e \equiv 0 \\
&\mathit{eval}\ (\mathit{If}\ c\ e1\ e2) &&= \texttt{if}\ \mathit{eval}\ c\ \texttt{then}\ \mathit{eval}\ e1\ \texttt{else}\ \mathit{eval}\ e2
\end{aligned}
$$

Evaluación usando GADTs: evaluador tagless

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ t \to t \\
&\mathit{eval}\ (\mathit{Val}\ n) &&= n \\
&\mathit{eval}\ (\mathit{Add}\ e1\ e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
&\mathit{eval}\ (\mathit{IsZero}\ e) &&= \mathit{eval}\ e \equiv 0 \\
&\mathit{eval}\ (\mathit{If}\ c\ e1\ e2) &&= \texttt{if}\ \mathit{eval}\ c\ \texttt{then}\ \mathit{eval}\ e1\ \texttt{else}\ \mathit{eval}\ e2
\end{aligned}
$$

- No hay posibilidad de fallos en tiempo de ejecución (salvo $\perp$)
- No se requieren tags
- El pattern matching sobre un GADT requiere signatura de tipo

GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

```
data Expr :: * -> * where
  Val    :: Int -> Expr Int
  Add    :: Expr Int -> Expr Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If     :: Expr Bool -> Expr t -> Expr t -> Expr t
  Pair   :: Expr a -> Expr b -> Expr (a, b)
  Fst    :: Expr (a, b) -> Expr a
  Snd    :: Expr (a, b) -> Expr b
```

GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

```
data Expr :: * → * where
  Val    :: Int → Expr Int
  Add    :: Expr Int → Expr Int → Expr Int
  IsZero :: Expr Int → Expr Bool
  If     :: Expr Bool → Expr t → Expr t → Expr t
  Pair   :: Expr a → Expr b → Expr (a, b)
  Fst    :: Expr (a, b) → Expr a
  Snd    :: Expr (a, b) → Expr b
```

Para *Fst* y *Snd* se esconde el tipo del componente no proyectado

GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

$$
\begin{aligned}
&\texttt{data}\ \mathit{Expr} :: * \to * \ \texttt{where}\\
&\quad \mathit{Val} \quad :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\
&\quad \mathit{Add} \quad :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\
&\quad \mathit{IsZero} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool}\\
&\quad \mathit{If} \quad :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ t \to \mathit{Expr}\ t \to \mathit{Expr}\ t\\
&\quad \mathit{Pair} \quad :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b)\\
&\quad \mathit{Fst} \quad :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a\\
&\quad \mathit{Snd} \quad :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ b
\end{aligned}
$$

Para *Fst* y *Snd* se esconde el tipo del componente no proyectado
Es como tener un tipo *existencial*:

$$\texttt{data}\ \mathit{Expr}\ a = \ldots \mid \forall\ b.\mathit{Fst}\ (\mathit{Expr}\ (a, b))$$

Ejemplo: Vectores

Un vector es una lista con largo:

```
data Vec a n where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Succ n)
```

Los números naturales los vamos a codificar como tipos vacíos:

```
data Zero
data Succ a
```

De esta forma, en el tipo del vector tenemos codificado su largo:

```
Nil                 :: Vec Int Zero
Cons 3 Nil          :: Vec Int (Succ Zero)
Cons 2 (Cons 3 Nil) :: Vec Int (Succ (Succ Zero))
```

head y tail

Las definiciones de *head* y *tail* son ahora seguras:

$$head :: Vec\ a\ (Succ\ n) \to a$$
$$head\ (Cons\ x\ xs) = x$$

$$tail :: Vec\ a\ (Succ\ n) \to Vec\ a\ n$$
$$tail\ (Cons\ x\ xs) = xs$$

El caso *Nil* es excluído porque no satisface el requerimiento de que la lista de entrada tenga largo mayor que cero.

Por lo tanto, las expresiones:

$$head\ Nil$$
$$tail\ Nil$$

resultan en un error de tipo.

Funciones sobre vectores

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ b\ n \\
&\mathit{map}\ f\ \mathit{Nil} = \mathit{Nil} \\
&\mathit{map}\ f\ (\mathit{Cons}\ x\ xs) = \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ xs)
\end{aligned}
$$

Funciones sobre vectores

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ b\ n \\
&\mathit{map}\ f\ \mathit{Nil} = \mathit{Nil} \\
&\mathit{map}\ f\ (\mathit{Cons}\ x\ xs) = \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{zipWith} :: (a \to b \to c) \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ b\ n \to \mathit{Vec}\ c\ n \\
&\mathit{zipWith}\ f\ \mathit{Nil}\ \mathit{Nil} = \mathit{Nil} \\
&\mathit{zipWith}\ f\ (\mathit{Cons}\ x\ xs)\ (\mathit{Cons}\ y\ ys) = \mathit{Cons}\ (f\ x\ y) \\
&\qquad\qquad (\mathit{zipWith}\ f\ xs\ ys)
\end{aligned}
$$

La función *zipWith* requiere que los vectores tengan el mismo largo

Funciones sobre vectores (2)

$$
\begin{aligned}
&\mathit{snoc} :: \mathit{Vec}\ a\ n \to a \to \mathit{Vec}\ a\ (\mathit{Succ}\ n) \\
&\mathit{snoc}\ \mathit{Nil}\ \quad\quad\quad\quad\quad y = \mathit{Cons}\ y\ \mathit{Nil} \\
&\mathit{snoc}\ (\mathit{Cons}\ x\ xs)\ y = \mathit{Cons}\ x\ (\mathit{snoc}\ xs\ y)
\end{aligned}
$$

Funciones sobre vectores (2)

$$
\begin{aligned}
&\mathit{snoc} :: \mathit{Vec}\ a\ n \to a \to \mathit{Vec}\ a\ (\mathit{Succ}\ n) \\
&\mathit{snoc}\ \mathit{Nil}\ \ \ \ \ \ \ \ \ \ \ \ \ y = \mathit{Cons}\ y\ \mathit{Nil} \\
&\mathit{snoc}\ (\mathit{Cons}\ x\ xs)\ y = \mathit{Cons}\ x\ (\mathit{snoc}\ xs\ y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse} :: \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ n \\
&\mathit{reverse}\ \mathit{Nil}\ \ \ \ \ \ \ \ \ \ \ \ = \mathit{Nil} \\
&\mathit{reverse}\ (\mathit{Cons}\ x\ xs) = \mathit{snoc}\ (\mathit{reverse}\ xs)\ x
\end{aligned}
$$

Concatenación de vectores

$$(+\!\!+) :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (m \oplus n)$$

Concatenación de vectores

$$(+\!\!+) :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (m \oplus n)$$

Podemos calcular $m \oplus n$ de la siguiente manera:

- construir evidencia explícita
- utilizar una type family (función a nivel de tipos)

Evidencia explícita

Codificar la suma como otro GADT:

```
data Sum m n s where
  SumZero :: Sum Zero n n
  SumSucc :: Sum m n s → Sum (Succ m) n (Succ s)

appV :: Sum m n s → Vec a m → Vec a n → Vec a s
appV SumZero      Nil         ys = ys
appV (SumSucc p) (Cons x xs) ys = Cons x (appV p xs ys)
```

Evidencia explícita

Codificar la suma como otro GADT:

```
data Sum m n s where
  SumZero :: Sum Zero n n
  SumSucc :: Sum m n s → Sum (Succ m) n (Succ s)

appV :: Sum m n s → Vec a m → Vec a n → Vec a s
appV SumZero      Nil         ys = ys
appV (SumSucc p) (Cons x xs) ys = Cons x (appV p xs ys)
```

Desventaja: tenemos que construir la evidencia a mano

Type family

```
type family (m :: *) :+: (n :: *) :: *
type instance Zero     :+: n = n
type instance (Succ m) :+: n = Succ (m :+: n)

(++) :: Vec a m -> Vec a n -> Vec a (m :+: n)
Nil       ++ ys = ys
Cons x xs ++ ys = Cons x (xs ++ ys)
```

Convertir entre listas y vectores

Sin problemas:

$$
\begin{aligned}
&\mathit{toList} :: \mathit{Vec}\ a\ n \to [a] \\
&\mathit{toList}\ \mathit{Nil} = [\,] \\
&\mathit{toList}\ (\mathit{Cons}\ x\ xs) = x : \mathit{toList}\ xs
\end{aligned}
$$

Convertir entre listas y vectores

Sin problemas:

$$
\begin{aligned}
&\mathit{toList} :: \mathit{Vec}\ a\ n \to [a] \\
&\mathit{toList}\ \mathit{Nil} = [\,] \\
&\mathit{toList}\ (\mathit{Cons}\ x\ xs) = x : \mathit{toList}\ xs
\end{aligned}
$$

No funciona:

$$
\begin{aligned}
&\mathit{fromList} :: [a] \to \mathit{Vec}\ a\ n \\
&\mathit{fromList}\ [\,] = \mathit{Nil} \\
&\mathit{fromList}\ (x : xs) = \mathit{Cons}\ x\ (\mathit{fromList}\ xs)
\end{aligned}
$$

Convertir entre listas y vectores

Sin problemas:

$$
\begin{aligned}
&\mathit{toList} :: \mathit{Vec}\ a\ n \to [a] \\
&\mathit{toList}\ \mathit{Nil} = [\,] \\
&\mathit{toList}\ (\mathit{Cons}\ x\ xs) = x : \mathit{toList}\ xs
\end{aligned}
$$

No funciona:

$$
\begin{aligned}
&\mathit{fromList} :: [a] \to \mathit{Vec}\ a\ n \\
&\mathit{fromList}\ [\,] = \mathit{Nil} \\
&\mathit{fromList}\ (x : xs) = \mathit{Cons}\ x\ (\mathit{fromList}\ xs)
\end{aligned}
$$

El tipo dice que el resultado tiene que ser polimórfico en n, pero no lo es.

De listas a vectores

Se puede:

- especificar el largo
- esconder el largo usando un tipo existencial

Especificando el largo

Los números naturales al nivel de los tipos los reflejamos al nivel de los valores usando un tipo **singleton**.

```
data SNat n where
  Zero :: SNat Zero
  Succ :: SNat n → SNat (Succ n)
```

SNat n tiene solo un valor por cada *n*:

```
Zero                :: SNat Zero
Succ Zero           :: SNat (Succ Zero)
Succ (Succ Zero)    :: SNat (Succ (Succ Zero))
```

Especificando el largo

Los números naturales al nivel de los tipos los reflejamos al nivel de los valores usando un tipo **singleton**.

```
data SNat n where
  Zero :: SNat Zero
  Succ :: SNat n → SNat (Succ n)
```

SNat n tiene solo un valor por cada *n*:

```
Zero                :: SNat Zero
Succ Zero           :: SNat (Succ Zero)
Succ (Succ Zero)    :: SNat (Succ (Succ Zero))
```

Conociendo el largo de antemano:

```
fromList :: SNat n → [a] → Vec a n
fromList Zero     []     = Nil
fromList (Succ n) (x : xs) = Cons x (fromList n xs)
fromList _        _      = error "wrong length!"
```

Usando existenciales

```
data VecAny a where
  VecAny :: Vec a n → VecAny a

fromList :: [a] → VecAny a
fromList []       = VecAny Nil
fromList (x : xs) = case fromList xs of
                      VecAny ys → VecAny (Cons x ys)
```

Usando existenciales

```
data VecAny a where
  VecAny :: Vec a n → VecAny a

fromList :: [a] → VecAny a
fromList []       = VecAny Nil
fromList (x : xs) = case fromList xs of
                      VecAny ys → VecAny (Cons x ys)
```

También podemos combinar ambas ideas e incluir un *SNat* en el tipo:

```
data VecAny a where
  VecAny :: SNat n → Vec a n → VecAny a
```

Reflexión de tipos

Mediante el uso de un GADT que refleje (represente) tipos:

```
data Type t where
  RInt  :: Type Int
  RChar :: Type Char
  RList :: Type a → Type [a]
  RPair :: Type a → Type b → Type (a, b)
```

es posible escribir funciones genéricas (type-indexed functions) de tipo

$$f :: Type\ a \to ...a\ ...$$

que hagan recursión en la representación de los tipos.

Ejemplo: función de compresión

Queremos comprimir valores de tipos representados en *Type*.

$$\texttt{data}\ \mathit{Bit} = 0 \mid 1$$

$$\begin{aligned}
&\mathit{compress} :: \mathit{Type}\ t \to t \to [\mathit{Bit}] \\
&\mathit{compress}\ (\mathit{RInt}) \quad\ \ i \quad\ \ = \mathit{compressInt}\ i \\
&\mathit{compress}\ (\mathit{RChar}) \quad\ \ c \quad\ \ = \mathit{compressChar}\ c \\
&\mathit{compress}\ (\mathit{RList}\ ra)\ \ [] \quad\ \ = 0 : [] \\
&\mathit{compress}\ (\mathit{RList}\ ra)\ \ (a : as) = 1 : \mathit{compress}\ ra\ a \\
&\qquad\qquad\qquad\qquad\qquad\qquad +\!\!+\ \mathit{compress}\ (\mathit{RList}\ ra)\ as \\
&\mathit{compress}\ (\mathit{RPair}\ ra\ rb)\ (a, b) = \mathit{compress}\ ra\ a +\!\!+ \mathit{compress}\ rb\ b
\end{aligned}$$

donde

$$\begin{aligned}
&\mathit{compressInt} \quad :: \mathit{Int} \to [\mathit{Bit}] \\
&\mathit{compressChar} :: \mathit{Char} \to [\mathit{Bit}]
\end{aligned}$$

son compresores para valores de *Int* y *Char*.