Programación Funcional Avanzada

Marcos Viera Alberto Pardo

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

Tipos de Datos Algebraicos Generalizados (GADTs)

Tipo de Datos Algebraico

$$\begin{aligned}\textbf{data}\ \mathit{Tree}\ a = {} & \mathit{Leaf} \\ & \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\end{aligned}$$

Introduce:

- un nuevo tipo de datos *Tree* de kind $* \rightarrow *$
- Constructores

$$\begin{aligned}\mathit{Leaf} &:: \mathit{Tree}\ a \\ \mathit{Node} &:: \mathit{Tree}\ a \rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a\end{aligned}$$

- la posibilidad de usar los constructores en patterns

Tipo de Datos Algebraico

$$\begin{aligned}\textbf{data}\ \mathit{Tree}\ a = {} & \mathit{Leaf} \\ & \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\end{aligned}$$

Introduce:

- un nuevo tipo de datos *Tree* de kind $* \to *$
- Constructores

$$\begin{aligned}\mathit{Leaf} &:: \mathit{Tree}\ a \\ \mathit{Node} &:: \mathit{Tree}\ a \to a \to \mathit{Tree}\ a \to \mathit{Tree}\ a\end{aligned}$$

- la posibilidad de usar los constructores en patterns

Syntaxis alternativa:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree} :: * \to * \ \textbf{where} \\ &\quad \mathit{Leaf} :: \mathit{Tree}\ a \\ &\quad \mathit{Node} :: \mathit{Tree}\ a \to a \to \mathit{Tree}\ a \to \mathit{Tree}\ a\end{aligned}$$

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

¿Tiene sentido levantar alguna de estas restricciones?

Ejemplo: Escribir un intérprete

$$
\begin{aligned}
&\textbf{data } \mathit{Expr} :: * \textbf{ where} \\
&\quad \mathit{Int} \quad :: \mathit{Int} \to \mathit{Expr} \\
&\quad \mathit{Bool} \quad :: \mathit{Bool} \to \mathit{Expr} \\
&\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
&\quad \mathit{Plus} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\quad \mathit{If} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{aligned}
$$

Por ejemplo, podemos escribir:

$$\mathit{If}\ (\mathit{IsZero}\ (\mathit{Plus}\ (\mathit{Int}\ 0)\ (\mathit{Int}\ 1)))\ (\mathit{Bool}\ \mathit{False})\ (\mathit{Bool}\ \mathit{True})$$

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

$$\textbf{if } \mathit{isZero}\ (0 + 1) \textbf{ then } \mathit{False} \textbf{ else } \mathit{True}$$

Evaluación

```
data Val :: * where
  VInt  :: Int  -> Val
  VBool :: Bool -> Val

eval :: Expr -> Val
eval (Int n)       = VInt m
eval (Bool b)      = VBool b
eval (IsZero e)    = case eval e of
                       VInt n -> VBool (n == 0)
                       _      -> error "type error"
eval (Plus e1 e2)  = case (eval e1, eval e2) of
                       (VInt n1, VInt n2) -> VInt (n1 + n2)
                       _                  -> error "type error"
eval (If e1 e2 e3) = case eval e1 of
                       VBool b -> if b then eval e2 else eval e3
                       _       -> error "type error"
```

Evaluación (2)

- El código de la evaluación se mezcla con el tratamiento de errores de tipos.
- El evaluador utiliza etiquetas ($VInt, VBool$) para distinguir valores - estas etiquetas se mantienen y chequean en tiempo de ejecución.

Evaluación (2)

- El código de la evaluación se mezcla con el tratamiento de errores de tipos.
- El evaluador utiliza etiquetas ($VInt$, $VBool$) para distinguir valores - estas etiquetas se mantienen y chequean en tiempo de ejecución.
- Podríamos escribir un type checker para prevenir errores de tipos en tiempo de ejecución.
  Pero no estaríamos usando el sistema de tipos de GHC para forzar esto.

Idea

Codificar el tipo del término que se representa en el propio tipo Haskell

$$
\begin{aligned}
&\textbf{data } \mathit{Expr} :: * \textbf{ where} \\
&\quad \mathit{Int} \quad :: \mathit{Int} \to \mathit{Expr} \\
&\quad \mathit{Bool} \quad :: \mathit{Bool} \to \mathit{Expr} \\
&\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
&\quad \mathit{Plus} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\quad \mathit{If} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{aligned}
$$

Idea

Codificar el tipo del término que se representa en el propio tipo Haskell

```
data Expr :: * where
  Int    :: Int → Expr
  Bool   :: Bool → Expr
  IsZero :: Expr → Expr
  Plus   :: Expr → Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

```
data Expr :: * → * where
  Int    :: Int → Expr Int
  Bool   :: Bool → Expr Bool
  IsZero :: Expr Int → Expr Bool
  Plus   :: Expr Int → Expr Int → Expr Int
  If     :: Expr Bool → Expr a → Expr a → Expr a
```

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 *Bool* o *IsZero*
  - cuando se analiza un *Expr Bool*, éste no puede ser construido por *Int* o *Plus*
  - cuando se analiza un *Expr Bool*, si encontramos *IsZero* en el pattern, sabemos que tenemos un *Expr Bool*
  - etc

Evaluación usando GADTs

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ a \to a \\
&\mathit{eval}\ (\mathit{Int}\ n) &&= n \\
&\mathit{eval}\ (\mathit{Bool}\ b) &&= b \\
&\mathit{eval}\ (\mathit{IsZero}\ e) &&= \mathit{eval}\ e == 0 \\
&\mathit{eval}\ (\mathit{Plus}\ e1\ e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
&\mathit{eval}\ (\mathit{If}\ e1\ e2\ e3) &&= \textbf{if}\ \mathit{eval}\ e1\ \textbf{then}\ \mathit{eval}\ e2\ \textbf{else}\ \mathit{eval}\ e3
\end{aligned}
$$

Evaluación usando GADTs

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ a \to a \\
&\mathit{eval}\ (\mathit{Int}\ n) &&= n \\
&\mathit{eval}\ (\mathit{Bool}\ b) &&= b \\
&\mathit{eval}\ (\mathit{IsZero}\ e) &&= \mathit{eval}\ e == 0 \\
&\mathit{eval}\ (\mathit{Plus}\ e1\ e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
&\mathit{eval}\ (\mathit{If}\ e1\ e2\ e3) &&= \textbf{if}\ \mathit{eval}\ e1\ \textbf{then}\ \mathit{eval}\ e2\ \textbf{else}\ \mathit{eval}\ e3
\end{aligned}
$$

- 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

Signaturas de tipo requeridas en GADTs

$$
\begin{aligned}
&\textbf{data } X :: * \to * \textbf{ where} \\
&\quad C :: Int \to X\ Int \\
&\quad D :: X\ a \\
&f\ (C\ n) = [n] \\
&f\ D\ \ \ = []
\end{aligned}
$$

¿Cual es el tipo de f?

Signaturas de tipo requeridas en GADTs

$$
\begin{aligned}
&\textbf{data } X :: * \to * \textbf{ where} \\
&\quad C :: Int \to X\ Int \\
&\quad D :: X\ a \\
&f\ (C\ n) = [n] \\
&f\ D\ \ \ = []
\end{aligned}
$$

¿Cual es el tipo de f?

$$
\begin{aligned}
&f :: X\ a \to [Int] \\
&f :: X\ a \to [a]
\end{aligned}
$$

Ninguno de los dos es instancia del otro

GADTs incluyen existenciales

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

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

GADTs incluyen existenciales

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

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

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

GADTs incluyen existenciales

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

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

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

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

Notación

Con kind explícito:

$$\begin{aligned}
&\textbf{data } \mathit{Expr} :: * \to * \textbf{ where} \\
&\quad \mathit{Int} \;\; :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
&\quad \mathit{Pair} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
&\quad \mathit{Fst} \;\; :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
&\quad \ldots
\end{aligned}$$

Notación

Con kind explícito:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr} :: * \to * \textbf{ where} \\
&\quad \mathit{Int} \;\; :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
&\quad \mathit{Pair} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
&\quad \mathit{Fst} \;\; :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
&\quad \ldots
\end{aligned}
$$

Con restricciones de tipo explícitas:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr}\ t \\
&\quad = \mathit{Int}\ \mathit{Int} && \mathit{with}\ t = \mathit{Int} \\
&\quad \mid \mathit{Pair}\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ b) && \mathit{with}\ t = (a, b) \\
&\quad \mid \mathit{Fst}\ (\mathit{Expr}\ (a, b)) && \mathit{with}\ t = a \\
&\quad \ldots
\end{aligned}
$$

Notación

Con kind explícito:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr} :: * \to * \textbf{ where} \\
&\quad \mathit{Int} \;\; :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
&\quad \mathit{Pair} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
&\quad \mathit{Fst} \;\; :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
&\quad \ldots
\end{aligned}
$$

Con restricciones de tipo explícitas:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr}\ t \\
&\quad = \mathit{Int}\ \mathit{Int} && \mathit{with}\ t = \mathit{Int} \\
&\quad \mid \mathit{Pair}\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ b) && \mathit{with}\ t = (a, b) \\
&\quad \mid \mathit{Fst}\ (\mathit{Expr}\ (a, b)) && \mathit{with}\ t = a \\
&\quad \ldots
\end{aligned}
$$

Notación de GHC:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr}\ t \textbf{ where} \\
&\quad \mathit{Int} \;\; :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
&\quad \mathit{Pair} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
&\quad \mathit{Fst} \;\; :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
&\quad \ldots
\end{aligned}
$$

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 tenemos codificados 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

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

Funciones sobre vectores

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

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

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

Funciones sobre vectores (2)

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

Funciones sobre vectores (2)

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

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

Funciones problemáticas

Append $(+\!\!+)$:

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

Funciones problemáticas

Append ($+\!\!+$):

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

¿Necesitamos funciones a nivel de tipos?

Funciones problemáticas

Append ($+\!\!+$):

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

¿Necesitamos funciones a nivel de tipos?

Convertir de listas a vectores:

$$fromList :: [a] \to Vec\ a\ n$$

Funciones problemáticas

Append ($+\!\!+$):

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

¿Necesitamos funciones a nivel de tipos?

Convertir de listas a vectores:

$$fromList :: [a] \to Vec\ a\ n$$

¿De dónde sale n?

Concatenación de vectores

Hay varias formas de resolver el problema:

- construir evidencia explícita
- utilizar type family

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

$$\textbf{type } \mathit{family}\ (m :: *)\ :+:\ (n :: *) :: *$$
$$\textbf{type instance } \mathit{Zero}\ :+:\ n = n$$
$$\textbf{type instance } (\mathit{Succ}\ m)\ :+:\ n = \mathit{Succ}\ (m\ :+:\ n)$$

$$(+\!\!+) :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (m\ :+:\ n)$$
$$\mathit{Nil} +\!\!+ \mathit{ys} = \mathit{ys}$$
$$\mathit{Cons}\ x\ \mathit{xs} +\!\!+ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{xs} +\!\!+ \mathit{ys})$$

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*.

$$\begin{aligned}&\textbf{data } \mathit{SNat}\ n \textbf{ where}\\ &\quad \mathit{Zero} :: \mathit{SNat}\ \mathit{Zero}\\ &\quad \mathit{Succ} :: \mathit{SNat}\ n \to \mathit{SNat}\ (\mathit{Succ}\ n)\end{aligned}$$

$\mathit{SNat}\ n$ tiene solo un valor por cada n:

$$\begin{aligned}&\mathit{Zero} &&:: \mathit{SNat}\ \mathit{Zero}\\ &\mathit{Succ}\ \mathit{Zero} &&:: \mathit{SNat}\ (\mathit{Succ}\ \mathit{Zero})\\ &\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}) &&:: \mathit{SNat}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))\end{aligned}$$

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
```

Comparar largo de vectores

Una opción:

$$equalLength :: Vec\ a\ m \rightarrow Vec\ a\ n \rightarrow Bool$$

Comparar largo de vectores

Una opción:

$$equalLength :: Vec\ a\ m \to Vec\ a\ n \to Bool$$

No muy útil, porque

if *equalLength v w*
then *head* (*zipWith* (,) *v w*)
else ...

no “tipa”.

Comparar largo de vectores

Una opción:

$$equalLength :: Vec\ a\ m \to Vec\ a\ n \to Bool$$

No muy útil, porque

if *equalLength v w*
then *head* (*zipWith* (,) *v w*)
else ...

no "tipa".

Como *equalLength* retorna sólo *Bool* se pierde la información de que m y n son iguales.

Tipo Igualdad

data *Equal a b* **where**
  Refl :: *Equal a a*

Tipo Igualdad

```
data Equal a b where
  Refl :: Equal a a
```

```
equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil          Nil          = Just Refl
equalLength (Cons _ xs) (Cons _ ys) =
                  case equalLength xs ys of
                    Just Refl        → Just Refl
                    Nothing          → Nothing
equalLength _            _           = Nothing
```

Tipo Igualdad

```
data Equal a b where
  Refl :: Equal a a
```

```
equalLength :: Vec a m → Vec b n → Maybe (Equal m n)
equalLength Nil          Nil          = Just Refl
equalLength (Cons _ xs) (Cons _ ys) =
                  case equalLength xs ys of
                    Just Refl        → Just Refl
                    Nothing          → Nothing
equalLength _            _           = Nothing
```

```
test :: Vec a m → Vec b (Succ n) → (a, b)
test v w = case equalLength v w of
             Just Refl → head (zipWith (,) v w)
```

Poder expresivo de la igualdad

El tipo *Equal* puede ser usado para codificar casi todos los demas GADTs:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr}\ t \textbf{ where} \\
&\quad \mathit{Int} \ :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
&\quad \mathit{If} \ :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ a \to \mathit{Expr}\ a \to \mathit{Expr}\ a \\
&\quad \mathit{Pair} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
&\quad \mathit{Fst} \ :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
&\quad \ldots
\end{aligned}
$$

se puede codificar como:

$$
\begin{aligned}
&\textbf{data } \mathit{Expr}\ t \\
&\quad = \quad\ \ \mathit{Int} \ \ (\mathit{Equal}\ t\ \mathit{Int}) \quad\ \ \mathit{Int} \\
&\quad \mid \quad\ \ \mathit{If} \qquad\qquad\qquad\ \ (\mathit{Expr}\ \mathit{Bool})\ (\mathit{Expr}\ t)\ (\mathit{Expr}\ t) \\
&\quad \mid \forall\ a\ b.\mathit{Pair}\ (\mathit{Equal}\ t\ (a, b))\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ b) \\
&\quad \mid \forall\ a\ b.\mathit{Fst} \ \ (\mathit{Equal}\ t\ a) \quad\ \ (\mathit{Expr}\ (a, b)) \\
&\quad \ldots
\end{aligned}
$$

Reflexión de tipos

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

$$
\begin{aligned}
&\textbf{data } \mathit{Type}\ t \textbf{ where} \\
&\quad \mathit{RInt} \quad :: \mathit{Type}\ \mathit{Int} \\
&\quad \mathit{RChar} :: \mathit{Type}\ \mathit{Char} \\
&\quad \mathit{RList} \ :: \mathit{Type}\ a \to \mathit{Type}\ [a] \\
&\quad \mathit{RPair} \ :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b)
\end{aligned}
$$

es posible escribir *funciones genéricas* de tipo

$$f :: \mathit{Type}\ a \to ...a\ ...$$

que hagan recursión en la representación de los tipos.

Ejemplo: función de compresión

Queremos comprimir valores de tipos representados en *Type*.

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

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

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*.

Valores Dinámicos

Se pueden definir valores dinámicos usando un GADT que "empaquete" una representación de tipo con un valor.

```
data Dynamic where
  Dyn :: Type a → a → Dynamic
```

Valores Dinámicos

Se pueden definir valores dinámicos usando un GADT que "empaquete" una representación de tipo con un valor.

```
data Dynamic where
  Dyn :: Type a → a → Dynamic
```

```
cast :: Dynamic → Type t → Maybe t
cast (Dyn ra a) rt = case tequal ra rt of
                       Just Refl → Just a
                       Nothing   → Nothing
```

donde:

```
tequal :: Type t → Type u → Maybe (Equal t u)
```

similar a *equalLength*.