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

$$\texttt{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

o alternativamente,

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

introduce:

- un nuevo tipo de datos *Tree* de kind $* \to *$
- 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
  - $T\ a_1\ ...\ a_n$ con $a_1, ..., a_n$ variables de tipo distintas

Restricciones

Los constructores de un tipo de datos T deben:

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

Vamos a levantar alguna de estas restricciones.

Deep embedding untyped

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \ \mathtt{where} \\
\quad \mathit{Val} \quad :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{If} \quad\ \ :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

Por ejemplo, podemos escribir:

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

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

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

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.

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \ \mathtt{where} \\
\quad \begin{array}{ll}
\mathit{Val} & :: \mathit{Int} \to \mathit{Expr} \\
\mathit{Add} & :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\mathit{IsZero} & :: \mathit{Expr} \to \mathit{Expr} \\
\mathit{If} & :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
\end{array}
$$

Deep embedding tipado

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

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \ \mathtt{where} \\
\quad \begin{array}{ll}
\mathit{Val} & :: \mathit{Int} \to \mathit{Expr} \\
\mathit{Add} & :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\mathit{IsZero} & :: \mathit{Expr} \to \mathit{Expr} \\
\mathit{If} & :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \to * \ \mathtt{where} \\
\quad \begin{array}{ll}
\mathit{Val} & :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{Add} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{IsZero} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ t \to \mathit{Expr}\ t \to \mathit{Expr}\ t
\end{array}
\end{array}
$$

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

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

Evaluación usando GADTs: evaluador tagless

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr}\ t \rightarrow t \\
\mathit{eval}\ (\mathit{Val}\ n) \qquad\ = n \\
\mathit{eval}\ (\mathit{Add}\ e1\ e2) = \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
\mathit{eval}\ (\mathit{IsZero}\ e) \quad = \mathit{eval}\ e \equiv 0 \\
\mathit{eval}\ (\mathit{If}\ c\ e1\ e2) = \mathtt{if}\ \mathit{eval}\ c\ \mathtt{then}\ \mathit{eval}\ e1\ \mathtt{else}\ \mathit{eval}\ e2
\end{array}
$$

- 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:

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

GADTs incluyen existenciales

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

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

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:

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

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

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

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:

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

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

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

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

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

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

Funciones sobre vectores (2)

$$
\begin{array}{l}
\mathit{snoc} :: \mathit{Vec}\ a\ n \rightarrow a \rightarrow \mathit{Vec}\ a\ (\mathit{Succ}\ n) \\
\mathit{snoc}\ \mathit{Nil}\ \qquad\qquad\ y = \mathit{Cons}\ y\ \mathit{Nil} \\
\mathit{snoc}\ (\mathit{Cons}\ x\ xs)\ y = \mathit{Cons}\ x\ (\mathit{snoc}\ xs\ y)
\end{array}
$$

Funciones sobre vectores (2)

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

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

Concatenación de vectores

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

Concatenación de vectores

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

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:

$$\texttt{data}\ \mathit{Sum}\ m\ n\ s\ \texttt{where}$$
$$\quad \mathit{SumZero} :: \mathit{Sum}\ \mathit{Zero}\ n\ n$$
$$\quad \mathit{SumSucc} :: \mathit{Sum}\ m\ n\ s \to \mathit{Sum}\ (\mathit{Succ}\ m)\ n\ (\mathit{Succ}\ s)$$

$$\mathit{appV} :: \mathit{Sum}\ m\ n\ s \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ s$$
$$\mathit{appV}\ \mathit{SumZero}\quad \mathit{Nil}\quad \mathit{ys} = \mathit{ys}$$
$$\mathit{appV}\ (\mathit{SumSucc}\ p)\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{appV}\ p\ \mathit{xs}\ \mathit{ys})$$

Evidencia explícita

Codificar la suma como otro GADT:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Sum}\ m\ n\ s\ \mathtt{where} \\
\quad \mathit{SumZero} :: \mathit{Sum}\ \mathit{Zero}\ n\ n \\
\quad \mathit{SumSucc} :: \mathit{Sum}\ m\ n\ s \to \mathit{Sum}\ (\mathit{Succ}\ m)\ n\ (\mathit{Succ}\ s)
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{appV} :: \mathit{Sum}\ m\ n\ s \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ s} \\
\mathit{appV}\ \mathit{SumZero} & \mathit{Nil} & \mathit{ys} & = \mathit{ys} \\
\mathit{appV}\ (\mathit{SumSucc}\ p) & (\mathit{Cons}\ x\ \mathit{xs}) & \mathit{ys} & = \mathit{Cons}\ x\ (\mathit{appV}\ p\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

Desventaja: tenemos que construir la evidencia a mano

Type family

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

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

Convertir entre listas y vectores

Sin problemas:

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

Convertir entre listas y vectores

Sin problemas:

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

No funciona:

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

Convertir entre listas y vectores

Sin problemas:

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

No funciona:

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

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

$$
\begin{array}{l}
\texttt{data}\ \mathit{VecAny}\ a\ \texttt{where} \\
\quad \mathit{VecAny} :: \mathit{Vec}\ a\ n \rightarrow \mathit{VecAny}\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{fromList} :: [a] \rightarrow \mathit{VecAny}\ a & & \\
\mathit{fromList}\ [\,] & = & \mathit{VecAny}\ \mathit{Nil} \\
\mathit{fromList}\ (x : xs) & = & \texttt{case}\ \mathit{fromList}\ xs\ \texttt{of} \\
 & & \quad \mathit{VecAny}\ ys \rightarrow \mathit{VecAny}\ (\mathit{Cons}\ x\ ys)
\end{array}
$$

$$
\begin{array}{l}
\mathtt{data}\ \mathit{VecAny}\ a\ \mathtt{where} \\
\quad \mathit{VecAny} :: \mathit{Vec}\ a\ n \rightarrow \mathit{VecAny}\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{fromList} :: [a] \rightarrow \mathit{VecAny}\ a & & \\
\mathit{fromList}\ [\,] & = & \mathit{VecAny}\ \mathit{Nil} \\
\mathit{fromList}\ (x : xs) & = & \mathtt{case}\ \mathit{fromList}\ xs\ \mathtt{of} \\
 & & \quad \mathit{VecAny}\ ys \rightarrow \mathit{VecAny}\ (\mathit{Cons}\ x\ ys)
\end{array}
$$

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

$$
\begin{array}{l}
\mathtt{data}\ \mathit{VecAny}\ a\ \mathtt{where} \\
\quad \mathit{VecAny} :: \mathit{SNat}\ n \rightarrow \mathit{Vec}\ a\ n \rightarrow \mathit{VecAny}\ a
\end{array}
$$

Reflexión de tipos

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

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

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

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

que hagan recursión en la representación de los tipos.

Ejemplo: función de compresión

Queremos comprimir valores de tipos representados en *Type*.

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

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

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*.