

**Informe del proyecto de grado:  
“Tipos dinámicos en lenguajes funcionales”**

**Carrera de Ingeniería en Computación  
Facultad de Ingeniería  
Universidad de la República**

Tutores: Alberto Pardo, Juan José Cabezas.  
Autor: Adrian Sieradzki

## Resumen

El objetivo de este proyecto es estudiar la extensión de un lenguaje funcional estáticamente tipado para admitir valores cuyo tipo es determinado dinámicamente. Dicha extensión se basa en la construcción de una biblioteca escrita en Haskell que utiliza tipos existenciales.

Las bibliotecas existentes de este tipo requieren que el usuario realice trabajo extra cada vez que se agrega una nueva definición de un tipo de dato. En este proyecto se apunta a automatizar ese proceso generando de manera automática el código que debería ingresar el usuario.

Se describe la construcción de un prototipo con estas características el cual tiene como objetivo comparar resultados y propiedades con otras extensiones existentes. Dicho prototipo fue implementado mediante el uso de Template Haskell. Se muestran diversos casos de estudio a fin de evaluar el impacto práctico que tiene la inclusión de tipos dinámicos en un lenguaje funcional.

Palabras clave: Dynamics, Generics, GADT, Template Haskell.

# Índice

1.	Introducción.....	4
2.	Conceptos básicos.....	6
2.1.	Tipos dinámicos.....	6
2.2.	Tipos existenciales.....	7
2.3.	Dynamics y Generics en Haskell.....	8
2.4.	Generalized Algebraic Data Types.....	15
2.5.	Template Haskell.....	16
3.	Descripción de la solución.....	18
3.1.	Implementación de Dynamics.....	18
3.2.	Generación de la representación genérica de tipos.....	20
3.3.	Uso de Dynamics con tipos de datos del usuario.....	23
3.4.	Diagrama.....	25
4.	Casos de estudio.....	26
4.1.	Recorrida de estructuras.....	26
4.2.	Listado de enteros de una estructura.....	27
4.3.	Cambio de valores en una estructura.....	30
5.	Conclusiones y trabajo futuro.....	35
5.1.	Conclusiones.....	35
5.2.	Limitaciones.....	35
5.3.	Dificultades.....	36
5.4.	Posibles extensiones.....	36
6.	Bibliografía.....	38

# 1. Introducción

Los lenguajes de programación con tipado estático realizan el chequeo de tipos en tiempo de compilación. De esta manera los posibles errores de tipo son detectados antes de que el programa sea ejecutado. Sin embargo existen situaciones en que puede ser necesario trabajar con valores cuyo tipo es dinámico, en el sentido que no puede ser determinado en tiempo de compilación. Esto ocurre, por ejemplo, con la manipulación de valores que son almacenados en memoria secundaria.

Las diversas propuestas de extensión de lenguajes funcionales estáticamente tipados con tipos dinámicos se basan en las ideas originalmente presentadas por Cardelli et. al. [1]. Esencialmente, todas ellas proponen la extensión del lenguaje funcional con un tipo universal, llamado *Dynamic*, en el cual los valores de tipo dinámico son embebidos. Los objetos de tipo *Dynamic* están formados por pares  $(v, \tau)$ , tal que  $v$  es un valor y  $\tau$  la representación del tipo de  $v$ . El lenguaje por lo tanto debe incluir mecanismos para construir e inspeccionar valores de tipo *Dynamic*.

Las extensiones con tipos dinámicos de lenguajes funcionales presentan diversas dificultades y restricciones tanto desde el punto de vista práctico como teórico. Sin embargo es importante destacar que la construcción de una extensión con tipos dinámicos permite ampliar el poder expresivo del lenguaje.

En este proyecto se desea experimentar con la construcción de una extensión basada en la propuesta presentada por Hinze y Cheney [3] la cual soporta tipos dinámicos y a la vez genéricos en Haskell.

Usando objetos de tipo *Dynamic* es posible expresar en efecto una importante clase de programas genéricos, como por ejemplo parsers o versiones genéricas de `printf` o `scanf`, cuyo comportamiento es uniforme para una familia de estructuras de datos. En este sentido este proyecto va a servir como laboratorio para analizar las ventajas y defectos que tienen los tipos dinámicos cuando son usados como mecanismo de programación genérica.

Tomemos como ejemplo la función `show` de Haskell. Dicha función sirve para mostrar como una cadena de caracteres (*String*) un valor de cualquier tipo. Para que dicha función se pueda aplicar a una determinada estructura de datos, se debe declarar dicha estructura como instancia de la clase *Show*, definiendo la función `show` para esa instancia. Esto es, cada estructura que desee tener una función `show` debe declararse por separado como instancia de la clase *Show*. Se asume que no se utiliza la cláusula *Deriving* al declarar el tipo de dato, aunque se desea que la función `show` funcione de la misma manera que si se utilizara dicha cláusula. La función `show` recorrerá la estructura del tipo mencionado, retornando strings acordes a lo encontrado. En caso que se desee recorrer una estructura de manera inversa al `show` estándar, se debe declarar una función específica para hacerlo o declarar una clase con dicha función y declarar como instancias de dicha clase a cada una de las estructuras que se deseen recorrer en forma inversa. En el correr de este trabajo mostraremos que cualquier estructura insertada en un tipo dinámico puede ser recorrida normalmente y a la inversa simplemente con escribir una sola vez la función necesaria para mostrar cualquier tipo dinámicos

Algunas de las técnicas a ser utilizadas en este proyecto ya han sido aplicadas en el diseño de una extensión de C desarrollada en el InCo, llamada C5 [2], la cual implementa una forma de *Dynamic*.

En este proyecto se desea comparar también el nivel de dificultad y las restricciones de una extensión basada en *Dynamic* con las características habituales de los lenguajes

funcionales con tipado estático. Se desea investigar el grado de complejidad y la expresividad de un lenguaje funcional extendido con tipos dinámicos. En este sentido, uno de los principales objetivos del proyecto es el diseño y construcción de un prototipo con estas características. Asimismo se desea experimentar con casos de estudio de relevancia práctica, en particular, los provenientes del área de programación genérica.

Este informe se encuentra organizado de la siguiente manera. La sección 2 presenta los conceptos básicos sobre los que se basa el presente trabajo. La sección 3 describe el aporte realizado por el proyecto. En la sección 4 se presentan ejemplos que utilizan las funcionalidades implementadas. Por último, en la sección 5 se presentan conclusiones y trabajos futuros.

## 2. Conceptos básicos

El presente proyecto se basa fuertemente en los conceptos de tipos dinámicos, tipos existenciales, GADTs y en el uso de Template Haskell para la implementación de un prototipo. En esta sección haremos una recorrida por cada uno de estos conceptos.

### 2.1. Tipos dinámicos

El trabajo de Cardelli et. al. [1] presenta el concepto de tipos dinámicos en lenguajes estáticamente tipados. La motivación planteada en ese trabajo indica que los lenguajes estáticamente tipados permiten la detección de errores durante la compilación (entre otras ventajas), pero sin embargo, incluso en estos lenguajes hay casos en los que el tipado no puede determinarse en tiempo de compilación.

Para resolver este problema, se propone extender dichos lenguajes con un nuevo tipo de dato llamado `Dynamic`, consistente en un par. Uno de los componentes de dicho par es el dato propiamente dicho, mientras que el otro componente contiene la información del tipo de dicho dato.

Esta extensión contará con una primitiva llamada `dynamic` (con d minúscula) la cual empaqueta un valor junto con su tipo en un `Dynamic`. También se contará con otra primitiva llamada `typecase`, la cual permite inspeccionar la etiqueta de tipo de un `Dynamic`.

Tomemos el siguiente ejemplo:

```
λx:Dynamic.  
  typecase x of  
    (i:Nat) i+1  
    else 0  
end
```

Si dicho ejemplo se aplica a `(dynamic 1:Nat)`, la expresión se evalúa retornando 2. Sin embargo, si el valor de tipo `Dynamic` es de tipo diferente a `Nat`, entonces la expresión retorna 0.

La sintaxis del `typecase` es la siguiente:

```
typecase esel of  
  ...  
  ( $\vec{x}_i$ ) ( $x_i:T_i$ ) ei  
  ...  
  else eelse  
end
```

donde  $e_{sel}$ ,  $e_i$ ,  $e_{else}$  son expresiones,  $x_i$  son variables,  $T_i$  son expresiones de tipo, y  $\vec{x}_i$  son listas de variables diferentes entre si. Las ocurrencias de variables en  $\vec{x}_i$  son ligantes y tienen alcance (scope) sobre  $T_i$  y  $e_i$ .

Para definir esta extensión formalmente, en [1] se presenta un cálculo lambda simplemente tipado con `Dynamic`, se dan las reglas de tipado y una semántica operacional. Se demuestra que si una expresión bien tipada  $e$  reduce a una expresión canónica  $v$ , entonces  $v$  tiene el mismo tipo que  $e$  (esta propiedad se conoce usualmente como “subject reduction”). De aquí se obtiene como corolario que ningún programa bien tipado puede realizar una evaluación errónea.

También se proponen extensiones a las ideas presentadas para lenguajes con polimorfismo implícito y explícito, tipos de datos abstractos y patrones de tipos más expresivos. En [9] se presentan dificultades encontradas al intentar integrar los tipos dinámicos en el lenguaje ML utilizando la propuesta de [1]. Estas dificultades se deben principalmente al tratamiento del polimorfismo dentro de los tipos dinámicos.

## 2.2. Tipos existenciales

Un ejemplo común para presentar los tipos existenciales es el del interés por contar con listas heterogéneas [8]. En los lenguajes funcionales, como Haskell, podemos definir un tipo de dato con un constructor distinto para cada uno de los tipos a aceptar en la lista. Por ejemplo:

```
data KEY = IntKey Int | BoolKey Bool | ListKey [Int]
```

De esta manera podemos tener una lista heterogénea:

```
hetList = [IntKey 5, ListKey [1,2,3], BoolKey True, IntKey 9]
```

Sin embargo, este enfoque tiene tres desventajas:

- Se debe recordar y usar muchos constructores diferentes.
- Los tipos componentes del tipo de datos algebraico no son abstractos, por lo tanto debo limitarme a utilizar funciones que operan solamente sobre los tipos definidos por los constructores. En el ejemplo estaría limitado a usar funciones que operen sobre los enteros, booleanos y listas de enteros.
- No se soportan extensiones del tipo. Si al tipo `KEY` se le agrega `FloatKey Float` entonces cada función que opera sobre el tipo se debe cambiar para incluir el nuevo caso.

Consideremos ahora la siguiente declaración de `KEY`:

```
data KEY a = Key a
```

En este caso el valor del constructor `Key` esta universalmente cuantificado según `a`:

```
Key :: a -> KEY a
```

Por otro lado, podríamos introducir una declaración que utiliza un tipo existencial `a`:

```
data KEY = Key a
```

Esta declaración es un nuevo tipo `KEY` sin parámetros con un constructor `Key` ahora de tipo:

```
Key :: a -> KEY
```

Como cualquier aplicación de `Key` es de tipo `KEY` podemos construir la siguiente lista:

```
hetList = [Key 5, Key [1,2,3], Key True]
```

A pesar que la lista es de tipo `[KEY]`, ella es heterogénea en el sentido de que sus elementos tienen diferentes tipos de representación. Más formalmente, diríamos

```
data KEY = forall a.Key a
```

y

```
Key :: forall a.a -> KEY.
```

Siendo que Haskell usa la palabra `forall` para describir existenciales. La mayoría de los compiladores de lenguajes funcionales soportan estos tipos existenciales.

Si quisiéramos por ejemplo utilizar funciones bien tipadas con los tipos componentes del tipo algebraico, podríamos por ejemplo utilizar definiciones de esta forma:

```
data KEY = forall a.Key a (a->Int)
```

Así tendríamos para el tipo de datos abstracto `a`, componente del tipo algebraico `KEY`, una función que toma dicho tipo de datos y retorna un `Int`.

### 2.3.Dynamics y Generics en Haskell

El trabajo de Hinze y Cheney [3] provee una implementación para trabajar con tipos dinámicos y genéricos utilizando Haskell estándar y tipos existenciales. El enfoque utilizado en dicho trabajo pretende ser más sencillo que el propuesto por Cardelli et al. [1] al no necesitar modificar el compilador ni realizar pruebas para demostrar que las expresiones bien tipadas no retornan evaluaciones erróneas. En esta sección revisamos dicho abordaje.

Comencemos asumiendo que tenemos una cierta familia de tipos y que deseamos definir una función (por ejemplo la igualdad) que funcione para todas los tipos de la familia (una función así se conoce como politépica [6]). Consideremos por ejemplo la familia de tipos dada por la siguiente gramática:

```
 $\tau :: \text{Int} \mid 1 \mid \tau + \tau \mid \tau \times \tau$ 
```

Tomaremos las siguientes declaraciones en Haskell para representar los tipos de la familia:

```
data 1 = Unit
data  $\alpha + \beta$  = Inl  $\alpha$  | Inr  $\beta$ 
data  $\alpha \times \beta$  =  $\alpha : \times : \beta$ 
```

Si quisiéramos definir la función politépica de igualdad para la familia de tipos, podemos empezar por remitirnos a la función de igualdad en Haskell cuyo tipo es:

```
(==) :: (Eq  $\alpha$ ) =>  $\alpha$  ->  $\alpha$  -> Bool
```

El contexto '`(Eq  $\alpha$ )`' explicita que la función sólo sirve para instancias de la clase `Eq`. La idea es sustituir el uso de las clases por un parámetro extra que representa el tipo para el cual se invoca la función de igualdad:

```
(==) :: Rep  $\alpha$  ->  $\alpha$  ->  $\alpha$  -> Bool
```



El valor de tipo  $\text{Rep } \tau$  corresponde a la representación del tipo  $\tau$ . El tipo  $\text{Rep}$  de las representaciones de tipo se puede definir de dos maneras, las cuales presentamos a continuación. La primera alternativa es definir, para cada constructor de tipo, un constructor correspondiente que represente dicho tipo. Para la familia de tipos definida anteriormente corresponderían los siguientes constructores:

```

 $R_{Int} :: \text{Rep } Int$ 
 $R_1 :: \text{Rep } 1$ 
 $R_+ :: \text{forall } \alpha. \text{Rep } \alpha \rightarrow (\text{forall } \beta. \text{Rep } \beta \rightarrow \text{Rep } (\alpha + \beta))$ 
 $R_\times :: \text{forall } \alpha. \text{Rep } \alpha \rightarrow (\text{forall } \beta. \text{Rep } \beta \rightarrow \text{Rep } (\alpha \times \beta))$ 

```

Por ejemplo, el tipo  $1 + (Int \times Int)$  es representado por el valor  $R_+ R_1 (R_\times R_{Int} R_{Int})$  de tipo  $\text{Rep } (1 + (Int \times Int))$ .

Para definir estos constructores utilizaremos declaraciones de tipos de datos de Haskell con alguna ligera extensión:

```

data Rep  $\tau$  =  $R_{Int}$  with  $\tau = Int$ 
              |  $R_1$  with  $\tau = 1$ 
              | forall  $\alpha \beta. R_+ (\text{Rep } \alpha) (\text{Rep } \beta)$  with  $\tau = \alpha + \beta$ 
              | forall  $\alpha \beta. R_\times (\text{Rep } \alpha) (\text{Rep } \beta)$  with  $\tau = \alpha \times \beta$ 

```

Esta extensión permite asignarle por ejemplo a  $R_{Int}$  el tipo  $\text{Rep } \tau$  cuando  $\tau = Int$ . Con esto ya podemos definir la igualdad politépica:

```

rEqual :: Rep  $\tau \rightarrow \tau \rightarrow \tau \rightarrow Bool$ 
rEqual ( $R_{Int}$ )  $t_1 t_2 = t_1 == t_2$ 
rEqual ( $R_1$ )  $t_1 t_2 = \text{case } (t_1, t_2) \text{ of}$ 
    ( $Unit, Unit$ )  $\rightarrow True$ 
rEqual ( $R_+ r_\alpha r_\beta$ )  $t_1 t_2 = \text{case } (t_1, t_2) \text{ of}$ 
    ( $Inl a_1, Inl a_2$ )  $\rightarrow rEqual r_\alpha a_1 a_2$ 
    ( $Inr b_1, Inr b_2$ )  $\rightarrow rEqual r_\beta b_1 b_2$ 
    _  $\rightarrow False$ 
rEqual ( $R_\times r_\alpha r_\beta$ )  $t_1 t_2 = \text{case } (t_1, t_2) \text{ of}$ 
    ( $a_1 : \times b_1, a_2 : \times b_2$ )  $\rightarrow$ 
        rEqual  $r_\alpha a_1 a_2 \ \&\& \ rEqual r_\beta b_1 b_2$ 

```

La no existencia de una cláusula `with` en las declaraciones de tipos de datos llevó a Cheney y Hinze a utilizar la alternativa que presentaremos a continuación.

Mediante la cláusula `with  $\alpha = \beta$`  queremos indicar que  $\alpha$  y  $\beta$  son equivalentes en algún sentido. Se propone entonces utilizar la equivalencia proposicional, definida como “sí y solo sí”.  $A \equiv B$  significa  $(A \Rightarrow B) \wedge (B \Rightarrow A)$ . Tomando las conjunciones como productos y los implica como funciones, podemos definir el tipo equivalencia  $\tau \leftrightarrow \tau'$  como  $(\tau \rightarrow \tau') \times (\tau' \rightarrow \tau)$ . Se pueden considerar las funciones componentes como traducciones de  $\tau$  a  $\tau'$  y viceversa. En Haskell declaramos:

```

data  $\alpha \leftrightarrow \beta = EP \{ from :: \alpha \rightarrow \beta, to :: \beta \rightarrow \alpha \}$ 

```

Un elemento de tipo  $\alpha \leftrightarrow \beta$  es una “prueba” de que los dos tipos son equivalentes.

Usando esta equivalencia de tipos definamos nuevamente  $\text{Rep}$ :

```

data Rep  $\tau$  =  $R_{Int}$  ( $\tau \leftrightarrow Int$ )
              |  $R_1$  ( $\tau \leftrightarrow 1$ )

```

```

| forall  $\alpha \beta$ .  $R_+$  (Rep  $\alpha$ ) (Rep  $\beta$ ) ( $\tau \leftrightarrow (\alpha + \beta)$ )
| forall  $\alpha \beta$ .  $R_\times$  (Rep  $\alpha$ ) (Rep  $\beta$ ) ( $\tau \leftrightarrow (\alpha \times \beta)$ )

```

Para concordar con los constructores  $R_\tau$  de la primera alternativa, presentamos los constructores inteligentes (smart constructors) que incorporan el valor de equivalencia reflexiva *self* como prueba de equivalencia.

```

self ::  $\alpha \leftrightarrow \alpha$ 
self = EP{from = id, to = id}
rInt :: Rep Int
rInt = RInt self
r1 :: Rep 1
r1 = R1 self
r+ :: Rep  $\alpha \rightarrow$  Rep  $\beta \rightarrow$  Rep ( $\alpha + \beta$ )
r+ r $\alpha$  r $\beta$  = R+ r $\alpha$  r $\beta$  self
r $\times$  :: Rep  $\alpha \rightarrow$  Rep  $\beta \rightarrow$  Rep ( $\alpha \times \beta$ )
r $\times$  r $\alpha$  r $\beta$  = R $\times$  r $\alpha$  r $\beta$  self

```

Veamos como se adapta la función de igualdad politépica:

```

rEqual :: Rep  $\tau \rightarrow \tau \rightarrow \tau \rightarrow$  Bool
rEqual (RInt ep) t1 t2 = from ep t1 == from ep t2
rEqual (R1 ep) t1 t2 = case (from ep t1 , from ep t2) of
    (Unit, Unit)  $\rightarrow$  True
rEqual (R+ r $\alpha$  r $\beta$  ep) t1 t2 = case (from ep t1 , from ep t2) of
    (Inl a1 , Inl a2)  $\rightarrow$  rEqual r $\alpha$  a1 a2
    (Inr b1 , Inr b2)  $\rightarrow$  rEqual r $\beta$  b1 b2
    _  $\rightarrow$  False
rEqual (R $\times$  r $\alpha$  r $\beta$  ep) t1 t2 = case (from ep t1 , from ep t2) of
    (a1 : $\times$ : b1 , a2 : $\times$ : b2)  $\rightarrow$ 
        rEqual r $\alpha$  a1 a2 && rEqual r $\beta$  b1 b2

```

Como podemos apreciar, cada vez que analizamos un valor *e* de tipo  $\tau$  equivalente a  $\tau'$ , reemplazamos *e* por *from ep e* donde *ep* ::  $\tau \leftrightarrow \tau'$  es la prueba de equivalencia correspondiente.

Si en lugar de retornar algo de tipo *Bool* retornáramos algo de un tipo  $\tau' = \tau$  deberíamos usar *to ep* en lugar de *from ep*. Por ejemplo:

```

rUno :: Rep  $\tau \rightarrow \tau$ 
rUno (RInt ep) = to ep (1 :: Int)

```

Al invocar una función politépica debemos proveer una representación como argumento. En Haskell se puede usar el sistema de clases para esta finalidad.

Definiremos la clase *Representable* de la siguiente manera:

```

class Representable  $\tau$  where
    rep :: Rep  $\tau$ 
instance Representable Int where
    rep = rInt
instance Representable 1 where
    rep = r1
instance (Representable  $\alpha$ , Representable  $\beta$ )  $\Rightarrow$ 

```

```

Representable ( $\alpha + \beta$ ) where
rep = r+ rep rep
instance (Representable  $\alpha$ , Representable  $\beta$ ) =>
Representable ( $\alpha \times \beta$ ) where
rep = rx rep rep

```

Podemos entonces definir una nueva función de igualdad politépica aprovechando esta definición:

```

cEqual :: (Representable  $\tau$ ) =>  $\tau \rightarrow \tau \rightarrow Bool$ 
cEqual t1 t2 = rEqual rep t1 t2

```

Con estas definiciones podemos ya definir tipos dinámicos como el par de valor y tipo del valor:

```

data Dynamic = forall  $\alpha$ . Dyn (Rep  $\alpha$ )  $\alpha$ 
dynamic :: (Representable  $\alpha$ ) =>  $\alpha \rightarrow Dynamic$ 
dynamic a = Dyn rep a

```

Para pasar un valor dinámico a su valor estático correspondiente o definir la función de igualdad sobre tipos dinámicos se debe confirmar que las representaciones coinciden.

La función *unify* toma dos representaciones de tipos y de ser posible retorna una prueba de su equivalencia.

```

unify :: Rep  $\tau_1 \rightarrow Rep \tau_2 \rightarrow Maybe (\tau_1 \leftrightarrow \tau_2)$ 

```

La unificación combina pruebas de equivalencia en una nueva prueba de equivalencia aplicando las leyes de simetría y transitividad. Estas leyes se corresponden con las funciones *inv* y  $\diamond$  respectivamente.

```

inv :: ( $\alpha \leftrightarrow \beta$ ) -> ( $\beta \leftrightarrow \alpha$ )
inv f = EP{from = to f, to = from f}
( $\diamond$ ) :: ( $\beta \leftrightarrow \gamma$ ) -> ( $\alpha \leftrightarrow \beta$ ) -> ( $\alpha \leftrightarrow \gamma$ )
f  $\diamond$  g = EP{from = from f . from g, to = to g . to f}

```

Para tipos paramétricos (como por ejemplo suma, producto, etc.) se deben tener un par de consideraciones. Por ejemplo, si a *unify* se le pasan las representaciones  $R_+ \ r_{\alpha_1} \ r_{\beta_1} \ ep_1 :: Rep \ \tau_1$  y  $R_+ \ r_{\alpha_2} \ r_{\beta_2} \ ep_2 :: Rep \ \tau_2$ , se tienen las pruebas  $ep_1 :: \tau_1 \leftrightarrow (\alpha_1 + \beta_1)$  y  $ep_2 :: \tau_2 \leftrightarrow (\alpha_2 + \beta_2)$ . Llamando recursivamente a *unify* podemos obtener pruebas de  $\alpha_1 \leftrightarrow \alpha_2$  y  $\beta_1 \leftrightarrow \beta_2$ . Usando la ley de congruencia podemos construir una prueba de  $(\alpha_1 + \beta_1) \leftrightarrow (\alpha_2 + \beta_2)$ . Finalmente las pruebas se combinan en una prueba de  $\tau_1 \leftrightarrow \tau_2$  aplicando simetría y transitividad. La ley de congruencia para la suma se corresponde con la función  $(\oplus)$ :

```

(+) :: ( $\alpha \rightarrow \beta$ ) -> ( $\gamma \rightarrow \delta$ ) -> (( $\alpha + \gamma$ ) -> ( $\beta + \delta$ ))
(f + g) (Inl a) = Inl (f a)
(f + g) (Inr b) = Inr (g b)
( $\oplus$ ) :: ( $\alpha \rightarrow \beta$ ) -> ( $\gamma \rightarrow \delta$ ) -> (( $\alpha + \gamma$ ) -> ( $\beta + \delta$ ))
f  $\oplus$  g = EP{from = from f + from g, to = to f + to g}

```

La función  $(\otimes)$  representa la ley de congruencia para pares y se define análogamente. Definimos entonces *unify*:

```

unify :: Rep  $\tau_1 \rightarrow \text{Rep } \tau_2 \rightarrow \text{Maybe}(\tau_1 \leftrightarrow \tau_2)$ 
unify  $r_1 r_2 = \text{case unify}' r_1 r_2 \text{ of}$ 
     $x : \_ \rightarrow \text{Just } x$ 
     $[] \rightarrow \text{Nothing}$ 
unify' :: Rep  $\tau_1 \rightarrow \text{Rep } \tau_2 \rightarrow [\tau_1 \leftrightarrow \tau_2]$ 
unify' ( $R_{\text{Int}} ep_1$ ) ( $R_{\text{Int}} ep_2$ )
    = [ $\text{inv } ep_2 \diamond ep_1$ ]
unify' ( $R_1 ep_1$ ) ( $R_1 ep_2$ )
    = [ $\text{inv } ep_2 \diamond ep_1$ ]
unify' ( $R_+ r_{\alpha 1} r_{\beta 1} ep_1$ ) ( $R_+ r_{\alpha 2} r_{\beta 2} ep_2$ )
    = [ $\text{inv } ep_2 \diamond (ep_{\alpha} \oplus ep_{\beta}) \diamond ep_1 \mid$ 
         $ep_{\alpha} \leftarrow \text{unify}' r_{\alpha 1} r_{\alpha 2}, ep_{\beta} \leftarrow \text{unify}' r_{\beta 1} r_{\beta 2}$ ]
unify' ( $R_{\times} r_{\alpha 1} r_{\beta 1} ep_1$ ) ( $R_{\times} r_{\alpha 2} r_{\beta 2} ep_2$ )
    = [ $\text{inv } ep_2 \diamond (ep_{\alpha} \otimes ep_{\beta}) \diamond ep_1 \mid$ 
         $ep_{\alpha} \leftarrow \text{unify}' r_{\alpha 1} r_{\alpha 2}, ep_{\beta} \leftarrow \text{unify}' r_{\beta 1} r_{\beta 2}$ ]
unify' _ _ = []

```

Usando esta unificación podemos pasar fácilmente un valor dinámico de un tipo representable a su valor estático.

```

cast :: (Representable  $\tau$ )  $\Rightarrow \text{Dynamic} \rightarrow \tau$ 
cast  $d = \text{rCast rep } d$ 
rCast :: Rep  $\tau \rightarrow \text{Dynamic} \rightarrow \tau$ 
rCast  $r_{\tau} (\text{Dyn } r_{\alpha} a) = \text{case unify } r_{\tau} r_{\alpha} \text{ of}$ 
     $\text{Just } ep \rightarrow \text{to } ep a$ 
     $\text{Nothing} \rightarrow \text{error "cast: type mismatch"}$ 

```

Nótese que el tipo de  $\tau$  no se puede inferir. Por lo tanto es obligatorio al invocar a `cast` hacer explícito el tipo de retorno. Por ejemplo, no podremos realizar `let a=dynamic('a') in cast a`, ya que no se conoce el tipo que debe retornar `cast`. En su lugar debemos realizar `let a=dynamic('a') in (cast a)::Char`.

Ahora si se puede definir la igualdad para tipos dinámicos, agregando el tipo `Dynamic` a la familia de tipos representables:

```

data Rep  $\tau = \dots$ 
    |  $R_{\text{Dynamic}} (\tau \leftrightarrow \text{Dynamic})$ 
rDynamic :: Rep Dynamic
rDynamic = RDynamic self
instance Representable Dynamic where
    rep = rDynamic
rEqual (RDynamic ep)  $d_1 d_2$ 
    = case (from ep  $d_1, \text{from } ep d_2$ ) of
        ( $\text{Dyn } r_{\alpha 1} a_1, \text{Dyn } r_{\alpha 2} a_2$ )  $\rightarrow$ 
            case unify  $r_{\alpha 1} r_{\alpha 2}$  of
                 $\text{Just } ep' \rightarrow \text{rEqual } r_{\alpha 1} a_1 (\text{to } ep' a_2)$ 
                 $\text{Nothing} \rightarrow \text{False}$ 

```

Con lo visto hasta ahora podemos definir funciones que sirven para todos los tipos representables. En [3] se propone ampliar el alcance de las funciones politípicas para que funcionen sobre todos los tipos. Ello lleva a que, para cada tipo que se desee manipular, el programador debe realizar algo de trabajo extra.

La idea es que todos los tipos sean representables. Tomemos por ejemplo los tipos de datos *Bool* y listas:

```
data Bool = False | True
data [α] = [] | α:[α]
```

Sus representaciones genéricas son:

```
Bool = 1+1
[α] = 1+(α×[α])
```

siendo los isomorfismos entre los tipos y sus respectivas representaciones genéricas los siguientes:

```
from_Bool :: Bool → 1+1
from_Bool False = Inl Unit
from_Bool True = Inr Unit
to_Bool :: 1+1 → Bool
to_Bool (Inl Unit) = False
to_Bool (Inr Unit) = True

from_[] :: [α] → 1+(α×[α])
from_[] [] = Inl Unit
from_[] (a:as) = Inr (a×as)
to_[] :: 1+(α×[α]) → [α]
to_[] (Inl Unit) = []
to_[] (Inr (a×as)) = a:as
```

Finalmente, las representaciones de estos tipos están dadas por:

```
r_Bool :: Rep Bool
r_Bool = R+ r1 r1 (EP from_Bool to_Bool)

r_[] :: Rep α → Rep [α]
r_[] r_α = R+ r1 (r× r_α (r_[] r_α)) (EP from_[] to_[])
```

Se puede apreciar que la representación de las listas está dada por un término infinito. La evaluación perezosa de Haskell hace que esto no sea un problema en algunas funciones politípicas. Sin embargo, esto no permite la unificación de representaciones y por ejemplo la evaluación de `cast (dynamic [0..9::Int]::[Int])` no finaliza. La solución propuesta es utilizar equivalencia por nombre y no estructural. Para ello se agrega el siguiente tipo de datos que alcanza para capturar términos de tipo cerrados (los constructores de tipo están aplicados a todos sus argumentos), incluyendo tipos de orden superior:

```
data Term = App String [Term] deriving (Eq)
```

Por ejemplo, `[Int]` es representado por `App "[]" [App "Int" []]`. Ahora tenemos el constructor adicional  $R_{Type}$  en el tipo *Rep*, el cual contiene un elemento de tipo *Term* para el nombre y otro de tipo *Rep*  $\alpha$  para la representación genérica.

```
data Rep τ = RInt (τ↔Int)
           | R1 (τ↔1)
```

```

| forall  $\alpha \beta$ .  $R_+$  (Rep  $\alpha$ ) (Rep  $\beta$ ) ( $\tau \leftrightarrow (\alpha + \beta)$ )
| forall  $\alpha \beta$ .  $R_\times$  (Rep  $\alpha$ ) (Rep  $\beta$ ) ( $\tau \leftrightarrow (\alpha \times \beta)$ )
|  $R_{dynamic}$  ( $\tau \leftrightarrow dynamic$ )
| forall  $\alpha$ .  $R_{type}$  Term (Rep  $\alpha$ ) ( $\tau \leftrightarrow \alpha$ )

```

Ahora tenemos dos formas de representación de tipos:  $Rep \tau$  captura la información estructural y  $Term$  captura la información de nombre. La función  $term_*$  extrae la información de nombre a partir de la información estructural.

```

term_* :: Rep  $\tau \rightarrow Term$ 
term_* (RInt ep) = App "Int" []
term_* (R1 ep) = App "1" []
term_* (R+ r $\alpha$  r $\beta$  ep) = App "(+)" [term_* r $\alpha$ , term_* r $\beta$ ]
term_* (R $\times$  r $\alpha$  r $\beta$  ep) = App "(*)" [term_* r $\alpha$ , term_* r $\beta$ ]
term_* (Rdynamic ep) = App "Dynamic" []
term_* (Rtype t r $\alpha$  ep) = t

```

Así cambiamos la definición de  $r_{[]}$  para incorporar el nombre del tipo lista.

```

r[] :: Rep  $\alpha \rightarrow Rep []$ 
r[] r $\alpha$  = Rtype (App "[]" [term_* r $\alpha$ ])
              (r+ r1 (r $\times$  r $\alpha$  (r[] r $\alpha$ ))) (EP from[] to[])

```

Queda extender la definición de  $unify'$  [3].

```

unify' (Rtype t1 r $\alpha_1$  ep1) (Rtype t2 r $\alpha_2$  ep2)
  = [inv ep2  $\diamond$  head (unify' r $\alpha_1$  r $\alpha_2$ )  $\diamond$  ep1 | t1 == t2]

```

Si  $t_1$  y  $t_2$  son iguales sabemos que  $r_{\alpha_1}$  y  $r_{\alpha_2}$  deben ser unificables.

Para ver el tratamiento de tipos de datos de alto orden veamos un ejemplo con el tipo de datos *generalized rose trees*.

```

data Tree  $\Phi \alpha$  = Node  $\alpha$  ( $\Phi$  (Tree  $\Phi \alpha$ ))

```

El argumento  $\Phi$  de *Tree* es un constructor de tipo de kind  $\ast \rightarrow \ast$ , por lo tanto *Tree* tiene kind  $(\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast$ . El constructor de la representación entonces tendrá tipo:

```

rTree :: (forall  $\alpha$ . Rep  $\alpha \rightarrow Rep$  ( $\Phi \alpha$ ))  $\rightarrow Rep \alpha \rightarrow Rep$  (Tree  $\Phi \alpha$ )

```

Queda definir los términos de tipo de las representaciones de  $\Phi$  y  $\alpha$ . Para averiguar el término de tipo de  $\Phi$  con tipo  $forall \alpha. Rep \alpha \rightarrow Rep (\Phi \alpha)$ , tomamos en cuenta que un término  $App s [t_1, \dots, t_n]$  aplicado a un término  $t$  retorna  $App s [t_1, \dots, t_n, t]$ . Entonces podemos reconstruir mediante aplicaciones la función original y su argumento utilizando la función *init* que dada una lista retorna dicha lista sin su último elemento.

```

term_ $\ast \rightarrow \ast$  :: (forall  $\alpha$ . Rep  $\alpha \rightarrow Rep$  ( $\Phi \alpha$ ))  $\rightarrow Term$ 
term_ $\ast \rightarrow \ast$  r $\Phi$  = case term_* (r $\Phi$  r1) of App t ts  $\rightarrow$  App t (init ts)

```

Con esto podemos definir  $r_{Tree}$ .

```

r_Tree r_φ r_α = R_Type (App "Tree" [term_{*→*} r_φ, term_* r_α])
                        (r_× r_α (r_φ (r_Tree r_φ r_α)))
                        (EP from_Tree to_Tree)

```

```

from_Tree :: Tree φ α → α × (φ (Tree φ α))
from_Tree (Node a ts) = (a :×: ts)
to_Tree :: α × (φ (Tree φ α)) → Tree φ α
to_Tree (a :×: ts) = Node a ts

```

Finalmente, en [3] se propone agregar los nombres de los constructores de un tipo de dato con el siguiente constructor en *Rep*.

```

data Rep τ = ...
            | R_Con String (Rep τ)

```

Así  $r_{Bool}$  por ejemplo quedaría:

```

r_Bool :: Rep Bool
r_Bool = R_Type (App "Bool" [])
            (R_+ (R_Con "False" r_1) (R_Con "True" r_1))
            (EP from_Bool to_Bool)

```

## 2.4. Generalized Algebraic Data Types

En [5] Hinze propone aplicaciones que usan declaraciones de tipo de la forma:

```

data Term t = Zero                                with t = Int
            | Succ (Term Int)                     with t = Int
            | Pred (Term Int)                     with t = Int
            | IsZero (Term Int)                   with t = Bool
            | If (Term Bool) (Term a) (Term a) with t = a

```

Dichas declaraciones pueden ser implementadas usando una extensión de Haskell, conocida como Generalized Algebraic Data Types (GADT) [12], que se puede encontrar en el compilador GHC de Haskell.

Por ejemplo, mediante el uso de GADTs se puede escribir la declaración del tipo *Term* como:

```

data Term a where
  Zero  :: Term Int
  Succ  :: Term Int -> Term Int
  Pred  :: Term Int -> Term Int
  IsZero :: Term Int -> Term Bool
  If     :: Term Bool -> Term a -> Term a -> Term a

```

Nótese que la cláusula `with` se reemplaza por la asignación de firmas a los constructores. De esta manera se especifican el tipo del constructor y de los elementos contenidos en dicho constructor. Por ejemplo, `Zero` es un constructor de tipo `Term Int`, `Succ` es un constructor que toma un elemento de tipo `Term Int` y es de tipo `Term Int`, etc.

El tipo `Term` es un ejemplo de lo que es conocido como un "phantom type" [5].

## 2.5.Template Haskell

Una herramienta fundamental en este trabajo es Template Haskell [11]. Template Haskell es una extensión a Haskell que permite meta-programar la generación de código en tiempo de compilación, esto es, permite la construcción algorítmica de programas en tiempo de compilación.

Template Haskell nos provee dos operaciones fundamentales: `reify` y `splice`. La primera permite obtener la representación interna de una función o tipo de datos en forma de una estructura de datos para manipularla como un valor en computaciones a ejecutarse en tiempo de compilación. En cambio, `splice` toma una estructura de datos (representación interna de un código) y la convierte en código Haskell.

Veamos un par de ejemplos, basados en los presentados en [11]. El primero pretende imitar en Haskell la función `printf` del lenguaje C. La manera de utilizar dicha función en Haskell es la siguiente:

```
$(printf "Error: %s on line %d") msg line
```

Aquí `msg` es de tipo `String` y `line` es de tipo `Int`. El operador `$` indica que se debe realizar una acción de “splice”. Asumamos que se tiene una función `parse` que parte un `String` en especificadores de formato, donde `D` significa presencia de `Int`, `S` presencia de `String`, y `L` presencia de texto.

```
data Format = D | S | L String
parse :: String -> [Format]
```

Por ejemplo, `parse "%d is %s"` retorna `[D, L " is ", S]`.

Si siempre se tuviera una lista con exactamente un especificador, se tendría:

```
printf :: String -> Expr
printf s = gen (parse s)
gen :: [Format] -> Expr
gen [D] = [| \n -> show n |]
gen [S] = [| \s -> s |]
gen [L s] = lift s
```

El código encerrado en los corchetes de tipo `[| _ |]` indica una acción de “quotation”, siendo retornado como representación interna de tipo expresión, lo cual en Template Haskell constituye un objeto de tipo `Expr`. Por su parte `lift` produce un elemento de tipo `Expr` que corresponde a la representación interna del argumento. Podríamos definir `gen` de otra manera. Por ejemplo, `gen [S] = [| \s -> s |]` puede ser escrito usando primitivas del tipo `Expr`:

```
gen [S] = lam [pvar "s"] (var "s")
```

Ambas declaraciones son equivalentes. Las declaraciones de `lam`, `pvar` y `var` son las siguientes:

```
lam ps e = do { e2 <- e; return(Lam ps e2) }
pvar = Pvar
var s = return(Var s)
```

Como se puede apreciar, `Expr` es un sinónimo para el tipo monádico `Q Exp` (ver [11]).



La declaración de `printf` de manera que `gen` sea recursiva para tratar con listas de especificadores es la siguiente.

```
printf :: String -> Expr
printf s = gen (parse s) [| "" |]

gen :: [Format] -> Expr -> Expr
gen [] x = x
gen (D : xs) x = [| \n-> $(gen xs [| $x++show n |]) |]
gen (S : xs) x = [| \s-> $(gen xs [| $x++s |]) |]
gen (L s : xs) x = gen xs [| $x ++ $(lift s) |]
```

Veamos ahora un ejemplo de `reify`, llamado aquí `reifyDecl`. Consideremos la siguiente definición de árbol binario.

```
data T a = Tip a | Fork (T a) (T a)
```

Al realizar `reifyDecl T`, obtenemos un valor del tipo `Q Dec`. Al analizar dentro de la mónada `Q` el valor de tipo `Dec`, veremos que contiene la representación interna de la declaración anterior:

```
Data "T" ["a"] [Constr "Tip" [Tvar "a"], Constr "Fork" [Tapp (Tcon
(Name "T")) (Tvar "a"),Tapp (Tcon (Name "T")) (Tvar "a"))]]
```

Por otro lado, podríamos realizar lo siguiente dentro de un módulo Haskell:

```
dec = Data "T" ["a"] [Constr "Tip" [Tvar "a"], Constr "Fork" [Tapp
(Tcon (Name "T")) (Tvar "a"),Tapp (Tcon (Name "T")) (Tvar "a"))]]
qdec :: Q Dec
qdec = return dec
$(qdec)
```

En este caso, al compilar el módulo se realizaría el splice de la declaración, es decir, el tipo de datos `T` quedaría declarado en este módulo. `dec` podría ser no solamente una declaración de tipo de datos, sino también una declaración de función, de clase o de instancia. Dicha declaración se evaluará en tiempo de compilación. Incluso `dec` podría aceptar parámetros que indiquen, por ejemplo, la declaración que se debe construir en función de los parámetros.

Template Haskell está implementado como una extensión de Haskell en GHC. La principal diferencia entre la versión actual y lo presentado en [11] (más allá de algunos cambios hechos en las estructuras de datos) es que en [11] figura como una posibilidad aplicar un splice a una función que esté definida en el mismo módulo que el splice. Esto no está permitido en la implementación de Template Haskell actualmente disponible. Si se desea realizar el splice de una estructura construida por una función, dicho splice debe ser hecho en un módulo distinto del módulo en el cual se define la función constructora.

### 3. Descripción de la solución

En la sección anterior se describieron conceptos basados en trabajos existentes. A partir de esta sección se describe el aporte hecho por el proyecto.

#### 3.1. Implementación de Dynamics

Las funciones `from` y `to` son introducidas originalmente en [3] debido a la falta de una cláusula `with` en las declaraciones de tipo en Haskell. De cualquier manera no podemos prescindir en la biblioteca de Dynamics del tipo de datos `↔` ni de las funciones `from` y `to`. Tanto Hinze en [4] como Oliveira y Gibbons en [10] explican la importancia para los tipos genéricos de las operaciones `to` y `from` que fueron presentadas en [3]. Para cada tipo de datos, dichas operaciones permiten transformar una instancia de un tipo de datos a una instancia de su representación genérica y viceversa. Esto permite utilizar los tipos dinámicos como tipos genéricos.

Nuestra implementación de tipos dinámicos se asemeja mucho a lo propuesto en [3] con la diferencia que se utilizan GADTs. Para ello se utilizan las extensiones de GHC que incluyen GADTs y tipos existenciales (`forall`).

Con la utilización de los GADT eliminamos la necesidad de incluir el operador de si y solo si (`↔`) en las representaciones de enteros, caracteres, unit, suma (que representamos usando el tipo de datos `Either`) y producto (que representamos usando el tipo de datos `par (a,b)`). Para el caso de la representación `RType` mantuvimos el si y solo si, necesario para transformar instancias de tipos de datos a instancias de sus representaciones genéricas y viceversa. Veremos un ejemplo de la utilidad de estas transformaciones en un ejemplo más adelante. Así queda la declaración de `Rep` en términos de GADT:

```
data Rep tau where
  RInt    :: Rep Int
  RFloat  :: Rep Float
  RChar   :: Rep Char
  RUnit   :: Rep ()
  RSum    :: Rep a -> Rep b -> Rep (Either a b)
  RProd   :: Rep a -> Rep b -> Rep (a,b)
  RFunc   :: Rep a -> Rep b -> Rep (a->b)
  RDyn    :: Rep Dynamic
  RType   :: Term -> Rep alpha -> (Iff tau alpha) -> Rep tau
  RCon    :: String -> Rep tau -> Rep tau
```

La otra diferencia principal que tenemos con [3] es la implementación de la función `cast` y sus funciones auxiliares. La función `cast` recibe un objeto de tipo dinámico y retorna el valor contenido en el mismo. Es importante notar que tanto en [3] como en nuestra implementación, el tipo de retorno de `cast` debe ser declarado en forma explícita al invocarse, como se indicó anteriormente. Empecemos por ver la signatura de `cast`:

```
cast :: Representable a => Dynamic -> a
```

Expandiendo la definición de `Dynamic`, vemos que en realidad `cast` es de tipo

```
cast :: Representable a => (Rep b) b -> a
```

Por lo tanto no nos podemos limitar a retornar el valor contenido en el tipo dinámico, ya que estaríamos retornando algo de tipo b cuando se espera algo de tipo a. Para resolver este asunto, aprovechamos el hecho de que el tipo a pertenece a la clase `Representable` y usamos la representación de a y de b para unificar a con b. Alcanza con ver la signature de la operación de unificación para ver el funcionamiento:

```
unify :: (Rep a) -> (Rep b) -> b -> Maybe a
```

Si ambas representaciones (de a y de b) son la misma, entonces podemos devolver el valor de tipo b dentro del valor `Just` (se unifican a y b). Si se devuelve `Nothing`, se devuelve una excepción de error en `cast`.

Veamos como quedan las funciones en nuestra biblioteca. La función `cast` no presenta cambios respecto a [3].

```
cast :: Representable a => Dynamic -> a
cast d = rCast rep d
```

La función `rCast` cambia con respecto a [3] en la medida en que la función `unify` cambia.

```
rCast :: (Rep tau) -> Dynamic -> tau
rCast rTau (Dyn rAlfa a) = case (unify rTau rAlfa a) of
    Just r -> r
    Nothing -> error "cast: type mismatch"
```

En este caso, `unify` no retorna una prueba de equivalencia, sino que toma dos representaciones y un valor del tipo representado por la segunda representación. Si ambas representaciones coinciden, retorna el valor de entrada dándole el tipo representado por la primera representación. En caso contrario retorna `Nothing`. Veamos la implementación de `cast`.

```
unify :: (Rep a) -> (Rep b) -> b -> Maybe a
unify RInt RInt val = Just val
unify RChar RChar val = Just val
unify RFloat RFloat val = Just val
unify RUnit RUnit val = Just val
unify RDyn RDyn val = Just val
unify (RSum ra rb) (RSum ral rbl) (Left va) =
    let u = (unify ra ral va)
    in if ((uniEq rb rbl)&&(uniEq ra ral))
        then Just (Left (fromJust u))
        else Nothing
unify (RSum ra rb) (RSum ral rbl) (Right vb) =
    let u = (unify rb rbl vb)
    in if ((uniEq rb rbl)&&(uniEq ra ral))
        then Just (Right (fromJust u))
        else Nothing
unify (RProd ra rb) (RProd ral rbl) (va,vb) =
    let u1 = (unify ra ral va)
        u2 = (unify rb rbl vb)
    in if ((not (isNothing u1))&&(not (isNothing u2)))
```

```

        then Just (fromJust u1, fromJust u2)
        else Nothing
unify r1@(RFunc ra rb) r2@(RFunc ral rbl) f =
  if ((uniEq ra ral)&&(uniEq rb rbl))
    then let faal = (\x -> fromJust (unify ral ra x))
          fblb = (\x -> fromJust (unify rb rbl x))
          in Just (fblb.f.faal)
    else Nothing
unify rt@(RType t ra ep) rtl@(RType tl ral epl) e =
  if (uniEq rt rtl)
    then let s1 = from epl e
          s2 = (unify ra ral s1)
          in Just (to ep (fromJust s2))
    else Nothing
unify (RCon s ra) (RCon s1 ral) t = unify ra ral t
unify _ _ _ = Nothing

```

La función auxiliar `uniEq` retorna `True` si ambas representaciones coinciden.

Para el caso en que ambas representaciones sean de números enteros, tanto `Rep a` como `Rep b` quedarán unificadas con `Rep Int` (es decir `a` y `b` quedarán unificadas con `Int`), por lo que se puede retornar el valor recibido tal y como se recibe.

Para las representaciones de tipo `Char`, `Float`, `Unit` y `Dynamic` la situación es igual que con el tipo `Int`.

En el caso de la suma, unificaremos el caso que corresponda y también verificaremos que para el otro sumando las representaciones se unifican.

En el caso del producto unificaremos ambos valores del producto usando sus respectivas representaciones.

En el caso de las representaciones de funciones verificaremos que los tipos que toman ambas funciones así como los tipos de retorno coinciden y retornaremos una nueva función que unifique los tipos que toma la función, le aplique la función al valor unificado y luego unifique el tipo de retorno.

En el caso en que se unifiquen constructores con nombre, ignoramos el nombre y unificamos el tipo.

Queda por analizar el caso en que se quieren unificar tipos definidos por el usuario, que llegarán representados por el constructor de representaciones `RType`. En este caso la unificación no es trivial. Aunque hacemos la comparación del nombre de las estructuras, como se propone en [3], queda por resolver la manera de indicarle a `unify` que algo del tipo `b`, es también del tipo `a`. Aquí es donde entra en juego la representación genérica de los tipos. Como se puede ver, definimos una variable `s1` que contenga la representación genérica del valor tomado (de tipo `b`) por `unify`. Luego unificamos esa representación con la otra representación (de tipo `a`). Como la variable que contiene la unificación será del tipo de la representación genérica de `a` (contenido dentro de un valor `Maybe`), podemos transformar dicha representación genérica al valor de tipo `a`. Entonces retornaremos ese valor sabiendo que su tipo es correcto.

### 3.2. Generación de la representación genérica de tipos

Como hemos visto, para utilizar un tipo de datos dentro de un tipo dinámico debemos declarar el tipo de datos como instancia de la clase `Representable`. Para ello necesitamos una función que construya la representación de dicho tipo de datos con los constructores definidos en el tipo de datos `Rep`. Para construir la representación de cualquier tipo de datos fuera de los enteros, los caracteres, la unidad, la suma y el

producto, se debe utilizar el constructor `RType` junto con el constructor `RCon` para cada uno de los constructores del tipo de datos. Notar que el constructor `RType` lleva como parámetro algo del tipo `si y solo si` (`Iff`) que hemos incluido con la finalidad de transformar el tipo de datos representado a una instancia de su representación genérica y viceversa (funciones `from` y `to` respectivamente).

Tanto en [3] como en los trabajos que utilizan el mismo enfoque, la tarea de escribir las declaraciones de instancias y de funciones `from`, `to` y de representación de tipo quedan a cargo del usuario. En éste trabajo proponemos automatizar la generación de estas declaraciones para tipos de datos no atómicos (por ejemplo, listas). Por tipos atómicos nos referimos a tipos de datos que no pueden ser definidos en función de otros tipos (por ejemplo: enteros, números de punto flotante, caracteres, etc.). El proceso de automatización debe también construir una representación del tipo de datos en términos del tipo `Rep`, lo que permite declarar como representable el tipo de datos y por lo tanto usar el constructor inteligente de tipos dinámicos para los valores del tipo mencionado (por ejemplo, construir un valor de tipo `Dynamic` a partir de una lista de enteros).

La generación de este código la realizamos utilizando la capacidad de Template Haskell para generar nuevas funciones en base a tipos de datos existentes. Como se ha visto, se puede obtener la representación interna de las declaraciones de tipos de datos. Con esta representación se puede deducir su representación genérica, y en base a la misma generar las declaraciones de las funciones `from`, `to`, la función constructora de representaciones para el tipo de dato y la declaración de instancia que declara el tipo de datos como instancia de `Representable`.

Actualmente contamos con una librería que genera las funcionalidades necesarias para los tipos de datos no atómicos, con algunas restricciones que detallaremos más adelante.

Con solo los nombres de los tipos de datos para los cuales queremos generar estas funcionalidades ya podemos, gracias a Template Haskell, obtener la información necesaria para realizar dicha generación. Con los nombres de los tipos de datos definidos por el usuario (y algunos definidos por Haskell como las listas), podemos producir las declaraciones en formato Template Haskell necesarias, por lo que recibiremos listas de nombres, y para cada uno de estos nombres se producirán las declaraciones de la función `to`, las declaraciones de la función `from`, las declaraciones del constructor de la representación y las declaraciones del tipo de datos como instancia de `Representable`.

Una vez que se tienen todas las declaraciones, se retorna la lista de declaraciones para que se pueda hacer el splice necesario, con el cual se reemplazará en tiempo de compilación la lista de declaraciones por el código correspondiente a dichas declaraciones.

Veamos un breve ejemplo. Tomemos el tipo de datos de las listas en Haskell. Solo con el nombre del tipo, es decir `[]`, podemos obtener su representación, cuya declaración en este caso tiene la forma `data [a] = [] | a:[a]`. Esta representación la obtendremos en forma de estructuras de Template Haskell. Una vez que tenemos esta representación, invocamos una serie de funciones que retornarán las declaraciones asociadas. Empecemos por ver la función que construye la signatura para la función `from`:

```
mkSgFrom :: Name -> [Con] -> [Name] -> [Dec]
```

Esta función retorna una lista de declaraciones y recibe el nombre del tipo de datos para el que se construirá la función `from`, la lista de constructores del tipo de datos y una lista con los nombres de las variables de tipo utilizadas en los tipos polimórficos. En nuestro ejemplo, la ultima lista contendría el nombre `a`. La lista de declaraciones retornada contiene en este caso una sola declaración, que cuando se le aplica el operador de splice se obtiene la siguiente declaración:

```
from[] :: [a] -> Either () (a,[a])
```

La función para construir la signatura de la función `to` es muy similar:

```
mkSgTo :: Name -> [Con] -> [Name] -> [Dec]
```

Para nuestro ejemplo, al realizar la acción de splice se obtendrá:

```
to[] :: Either () (a,[a]) -> [a]
```

De la misma manera las funciones `mkFrom :: Name -> [Con] -> [Dec]` y `mkTo :: Name -> [Con] -> [Dec]` construirán las siguientes declaraciones para nuestro ejemplo:

```
from[] [] = Left ()
from[] ((:) a1 a2) = Right (a1,a2)
```

```
to[] (Left ()) = []
to[] (Right (a1,a2)) = (:) a1 a2
```

Veamos ahora las funciones para construir la representación del tipo de datos y la signatura de dicha función. Estas son respectivamente

`mkSgReps :: Name -> [Con] -> [Name] -> Q [Dec]` y `mkReps :: Name -> [Con] -> [Name] -> [Dec]`. Para nuestro ejemplo, estas funciones construirán las siguientes declaraciones:

```
r[] :: Rep a -> Rep [a]
```

```
r[] ra = RType (App "[]" [termS ra])
              (RSum (RCon "[]" RUnit) (RCon ":" (RProd ra (r[] ra))))
              (EP from[] to[])
```

Queda por ver una última función la cual construirá la declaración de un tipo de datos como instancia de la clase `Representable`:

```
mkInst :: Name -> [Con] -> [Name] -> [Dec]
```

En nuestro ejemplo, obtendremos la siguiente declaración:

```
instance (Representable a) => Representable ([] a) where
    rep = r[] rep
```

En [3], Cheney y Hinze hablan del tratamiento para tipos de datos que incluyen tipos de rango 2. Como ejemplo utilizan los rose trees generalizados explicando cómo construir las funciones `from`, `to` y la función que retorna la representación de los rose trees generalizados. Veamos este ejemplo nuevamente:

```
data Tree  $\Phi$   $\alpha$  = Node  $\alpha$  ( $\Phi$  (Tree  $\Phi$   $\alpha$ ))
```

```

rTree :: (forall  $\alpha$ . Rep  $\alpha \rightarrow \text{Rep } (\Phi \alpha) \rightarrow \text{Rep } \alpha \rightarrow \text{Rep } (\text{Tree } \Phi \alpha)$ )
rTree r $\Phi$  r $\alpha$  = RType (App "Tree" [term $_{* \rightarrow *}$  r $\Phi$ , term $_*$  r $\alpha$ ])
              (RCon "Node" (r $_*$  r $\alpha$  (r $\Phi$  (rTree r $\Phi$  r $\alpha$ ))))
              (EP fromTree toTree)

fromTree :: Tree  $\Phi \alpha \rightarrow \alpha \times (\Phi (\text{Tree } \Phi \alpha))$ 
fromTree (Node a ts) = (a :x: ts)
toTree ::  $\alpha \times (\Phi (\text{Tree } \Phi \alpha)) \rightarrow \text{Tree } \Phi \alpha$ 
toTree (a :x: ts) = Node a ts

```

Sin embargo, en [3] no se explica cómo declarar estos tipos de datos como representables (condición necesaria y suficiente para utilizar los constructores inteligentes de la librería de tipos dinámicos con el tipo de datos en cuestión). Para realizar esta declaración no se puede dejar sin especificar el tipo concreto del tipo de rango 2. Por ejemplo, en el caso de los rose trees generalizados se debería declarar como representables los rose trees generalizados que utilicen el constructor de listas en el constructor de tipo de rango 2 (de esta manera serían representables los rose trees comunes):

```

instance (Representable  $\alpha$ ) => Representable (Tree []  $\alpha$ ) where
    rep = rTree r[] rep

```

La solución que encontramos para este dilema es la de ponerle como requisito al usuario que declare como `type` el tipo de datos que incluye constructores de rango 2, y que especifique qué constructores de rango 2 usará.

Por ejemplo, en el caso de los rose trees generalizados, se debería declarar como `type` a los rose trees generalizados que utilizan el constructor de listas para que el constructor inteligente de tipos dinámicos acepte rose trees comunes. Veamos la declaración:

```

type Rtree  $\alpha$  = Tree ([])  $\alpha$ 

```

Como se comentó anteriormente, debido a la manera en que se deben tratar los tipos de datos con parámetros de rango 2, se cuenta también con una segunda lista nombres, que recoge las declaraciones de tipo `type` que especifican qué constructores de rango 2 se utilizarán.

También se decidió retornar un mensaje de error si se intenta construir la representación de algún tipo de datos con parámetros de rango mayor que 2, ya que estos tipos no son soportados por la biblioteca de tipos dinámicos. Una extensión a futuro es incorporar el tratamiento de tipos de rango arbitrario.

### 3.3. Uso de Dynamics con tipos de datos del usuario

Para simplificar aún más la tarea del usuario, se implementó la generación de un módulo `A_Decs` a partir de un módulo `A` escrito en Haskell 98. Dicho módulo contiene una lista con todos los tipos definidos por el usuario en `A` mediante una declaración `data` y otra lista con todos los tipos definidos mediante `type`. También importa el módulo `Mkdecs` y con las dos listas anteriores invoca la funcionalidad que realiza los reify necesarios y produce la lista de declaraciones necesarias para usar la librería de

tipos dinámicos con los tipos declarados en el módulo A. Finalmente se encarga de hacer el splice de las declaraciones. Por lo tanto, si se quiere en otro módulo usar el constructor inteligente de tipos dinámicos o las funciones `from` o `to` con un tipo de datos declarado en A, se debe importar el módulo `A_Decs`.

Estos aportes nos acercan a la construcción de una extensión para Haskell que permita al usuario manejar tipos dinámicos sin preocuparse por realizar tareas adicionales (como escribir las funciones `from` y `to`).

Veamos un ejemplo. Supongamos que tenemos el siguiente módulo:

```
module Structs where

data BTree a = Leaf a | BNode (BTree a) (BTree a)
data Tree fi alfa = Node alfa (fi (Tree fi alfa))
type RoseT a = Tree ([]) a
```

Usando la utilidad descrita en este módulo, se generará el siguiente módulo:

```
{- CODIGO GENERADO -}
module Structs_Decs where
import Language.Haskell.TH
import Dynamics
import Mkdecs
import Basic_Decs
import Structs

$(let l1 = [("BTree", "Structs"), ("Tree", "Structs")]
    l2 = [("RoseT", "Structs")]
    in constL l1 l2)
```

La declaración parentizada indica que se deben generar las declaraciones de funciones `from`, `to`, función de representación de los tipos `BTree` y `Tree` y la declaración de dichos tipos como instancias de `Representable`. Al tener `Tree` un parámetro de rango 2, no se generará la declaración de instancia de `Representable` para `Tree`. Sin embargo, como se tiene en la segunda lista el nombre del tipo `RoseT`, se generará la declaración de instancia de `Representable` para las instancias de `Tree` que utilicen listas en el parámetro de rango 2 de `Tree`.

Dichas declaraciones están expresadas en la representación de Template Haskell. El operador de splice (\$) indica que en tiempo de compilación esas expresiones de Template Haskell se deben reemplazar por el código Haskell que representan. Por lo tanto, si se desea utilizar en un tercer módulo el tipo de datos `BTree` o `Tree` como tipos dinámicos, basta con importar los módulos `Structs` y `Structs_Decs`.

Queda aclarar que el módulo `Basic_Decs` contiene las declaraciones de instancias de la clase `Representable` para algunos tipos de datos definidos en el prelude. Su estructura podría ser la siguiente:

```
module Basic_Decs where
import Language.Haskell.TH
import Dynamics
import Mkdecs

$(let l1 = [("Bool", "GHC.Base"),
```



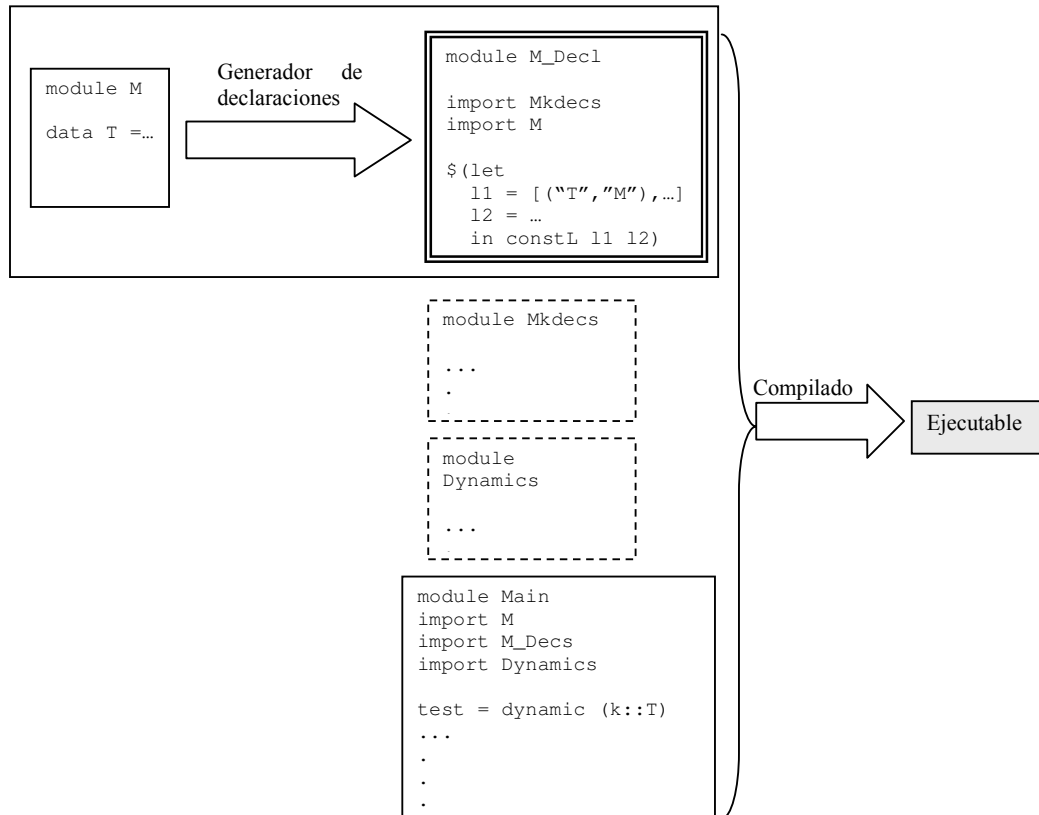
```

(["[]", "GHC.Base"), ("Maybe", "Data.Maybe")]
12 = []
in constL 11 12)

```

### 3.4.Diagrama

A continuación presentamos un diagrama de cómo funciona el proceso de generación. Los cuadros normales muestran código de usuario, los cuadros de borde punteado son las acciones asociadas a las subsecciones 3.1 a 3.3, los cuadros de borde doble son los módulos generados y los cuadros con relleno son ejecutables generados.



## 4. Casos de estudio

En esta sección presentaremos ejemplos que muestran las capacidades de los tipos dinámicos. El primero permite mostrar al derecho o a la inversa cualquier estructura que se pueda guardar en un tipo dinámico. El segundo ejemplo busca todas las ocurrencias de números enteros en un tipo dinámico y los retorna en una lista. El tercer ejemplo toma una estructura que representa una compañía y aumenta un 10% el sueldo de sus empleados. Veremos como realizar el segundo y tercer ejemplo de dos maneras distintas, tal que una de las maneras es similar al enfoque utilizado en [7] y analizaremos las ventajas y desventajas de cada una de ellas.

### 4.1. Recorrida de estructuras

Se desea mostrar al derecho o a la inversa cualquier estructura que se pueda guardar en un tipo dinámico.

Veamos la definición de esta función de Show politépica [6]:

```
pShow :: Representable a => a -> Bool -> String
pShow a b = pShow' (dynamic a) b Nothing
```

La función lleva como parámetros el tipo propiamente dicho y un booleano que indica si la estructura se mostrará en forma inversa o no. Veamos ahora la función auxiliar:

pShow'.

```
pShow' :: Dynamic -> Bool -> Maybe Term -> String
pShow' (Dyn rb b) bool t = case rb of
```

Analicemos caso por caso según el tipo de dato contenido en el valor dinámico.

Para el caso que el tipo dinámico contenga un entero, un real o un carácter, el mismo se retorna como string.

```
RInt -> show b
RFloat -> show b
RChar -> [b]
```

Si el tipo dinámico contiene una suma se recorre recursivamente el caso que corresponda.

```
RSum rl rr -> case b of
  Left l -> pShow' (Dyn rl l) bool t
  Right r -> pShow' (Dyn rr r) bool t
```

En el caso de que el contenido del valor dinámico sea una función no tiene sentido desplegarlo en pantalla, por lo que se mostrará un mensaje de error. En el caso de que el contenido del valor dinámico sea de tipo dinámico, se recorrerá recursivamente dicho contenido.

```
RFunc rf1 rf2 -> error "function found"
RDyn -> "(Dynamic " ++ (pShow' b bool t) ++ ")"
```

En el próximo caso se ve el porque del parámetro de tipo `Maybe Term` de la función. Al principio no se desea patentizar una estructura. Sin embargo, si estamos recorriendo una estructura y encontramos otra estructura, desearíamos patentizar la nueva estructura.

```
RType term ra ep -> if (isNothing t)
  then pShow' (Dyn ra (from ep b)) bool (Just term)
  else "(" ++ pShow' (Dyn ra (from ep b)) bool (Just term) ++ ")"
```

Cuando se encuentran nombres de constructores, se evalúa si son operadores o no. En caso de no serlo, se imprime el nombre del constructor y se continúa la recorrida de la estructura. Si es un operador, se patentiza y se continúa con la recorrida.

```
RCon s ra' -> if (not $ isOperator s)
  then s ++ " " ++ (pShow' (Dyn ra' b) bool t)
  else "(" ++ s ++ " " ++ (pShow' (Dyn ra' b) bool t)
```

Cuando se encuentra algo de tipo unidad, se retorna el string vacío.

```
RUnit -> ""
```

En caso de encontrar algo de tipo producto, se realiza un análisis de casos. Si se está mostrando la estructura invertida, primero se mostrará el valor de la derecha y luego el de la izquierda. Si se esta mostrando la estructura al derecho, se muestra primero el valor de la izquierda y luego el de la derecha.

```
RProd rp1 rp2 -> case b of
  (p1,p2) -> case bool of
    True ->
      (pShow' (Dyn rp2 p2) bool t)
      ++
      (pShow' (Dyn rp1 p1) bool t)
    False ->
      (pShow' (Dyn rp1 p1) bool t)
      ++
      (pShow' (Dyn rp2 p2) bool t)
```

Por ejemplo, al invocar la función

```
putStrLn $ pShow
  (BTreeNode (Leaf (1::Int)) (BTreeNode (Leaf 2) (Leaf 3)))
  True
```

obtendremos la siguiente salida:

```
((BTreeNode ((BTreeNode ((Leaf (3)) (Leaf (2)))) (Leaf (1)))))
```

Se ve aquí el árbol binario invertido.

Se puede sustituir el árbol binario de enteros por cualquier otra estructura que se pueda guardar en un valor dinámico.

## ***4.2.Listado de enteros de una estructura***

En este ejemplo buscaremos dentro de una estructura todas las ocurrencias de números enteros y los retornaremos en una lista.

Veamos la definición de dicha función.

```
getInts :: Representable a => a -> [Int]
getInts a = aux (dynamic a)
  where
    aux :: Dynamic -> [Int]
    aux (Dyn ra a) = case ra of
      RInt -> [a]
      RFloat -> []
      RChar -> []
      RUnit -> []
      RSum rl rr -> case a of
        Left l -> aux (Dyn rl l)
        Right r -> aux (Dyn rr r)
      RProd rp1 rp2 -> case a of
        (p1,p2) -> (aux (Dyn rp1 p1)) ++ (aux (Dyn rp2 p2))
      RFunc rf1 rf2 -> error "function found"
      RDyn -> aux a
      RType term ra' ep -> aux (Dyn ra' (from ep a))
      RCon s ra' -> aux (Dyn ra' a)
```

Como se puede apreciar, la estructura es almacenada en un tipo dinámico y luego se evalúa la representación de tipo de esa estructura para definir la acción a tomar. Si el tipo es entero, se retorna la lista con dicho entero. Si la estructura es de tipo real, carácter o unidad, se retorna la lista vacía. Si la estructura es de tipo suma, se retorna la lista de enteros que se obtiene de realizar la búsqueda recursiva de enteros en el caso que corresponda. Si la estructura es de tipo producto, se concatenan las listas que se obtienen aplicándole la búsqueda recursiva a cada una de las componentes del producto. Si la estructura es una función, no tiene sentido buscar enteros, por lo que se retornará un mensaje de error. Si la estructura es de tipo `Dynamic` se realiza la búsqueda de manera recursiva en dicho tipo dinámico. Si la estructura es de otro tipo de datos, se realiza la búsqueda de enteros de manera recursiva en la representación genérica de dicho tipo de datos. Por último, si el tipo de datos es un constructor, se buscan ocurrencias de números enteros de manera recursiva en el contenido de dichos constructores.

Otra manera de obtener esta lista de enteros a partir de una estructura está basada en [7]. Veamos como funciona.

```
wrapInt :: Int -> [Int]
wrapInt i = [i]

getInts' :: Representable a => a -> [Int]
getInts' a = everything [] (++) (wrapInt) a
```

Tenemos una función que dado un entero retorna una lista que lo contiene. Luego presentamos otra función que utiliza la función `everything`. Dicha función toma un valor por defecto del tipo de retorno, una función que combina dos valores del tipo de retorno, una función que toma un valor de un tipo determinado y retorna un elemento del tipo de retorno y por ultimo toma una estructura. De esta forma, `everything` aplica

la función dada (en este caso `wrapInt`) a todos los elementos de la estructura con una estrategia top-down y left-right. En caso de no corresponder el tipo del valor al que se le aplica la función dada con el tipo esperado por dicha función, se retornará el valor por defecto. A medida que se vayan obteniendo valores de retorno, los mismos se combinan utilizando el combinador (en este caso `++`). Antes de ver la implementación de `everything`, veamos la implementación de `apply'`. Esta función toma tres valores dinámicos. El primero contiene un valor de tipo `b`, el segundo una función de tipo `a->b` y el tercero contiene un valor de tipo `a'`. En caso de que sea posible unificar los tipos `a` y `a'`, se le aplica la función de tipo `a->b` al valor de tipo `a'` (unificado con `a`) y se retorna el resultado de esta aplicación, en forma de un valor dinámico de tipo `b`. En caso de no poder unificar los tipos `a` y `a'`, se retorna el valor dinámico de tipo `b` tomado como parámetro. Notar que en cualquier caso el tipo de retorno es `b`.

```

apply' :: Dynamic -> Dynamic -> Dynamic -> Dynamic
apply' db@(Dyn rb' b) df@(Dyn (RFunc ra rb) f) da@(Dyn ra' a) =
  if (uniEq rb rb')
    then (if (uniEq ra ra')
          then apply df da
          else db)
    else error "Default value's type does not match return type"

apply :: Dynamic -> Dynamic -> Dynamic
apply (Dyn (RFunc ralfa rbeta) f) (Dyn ralfa' x) =
  case (unify ralfa ralfa' x) of
    Just x' -> (Dyn rbeta (f x'))
    Nothing -> error "apply: type mismatch"
apply _ _ = error "apply: not a function"

```

La función `apply` ya estaba definida en [3] y se encuentra en la biblioteca de `Dynamics`.

Ahora si podemos definir `everything`.

```

everything :: (Representable a, Representable b, Representable r) =>
  r -> (r -> r -> r) -> (b -> r) -> a -> r
everything def combinator f struct = aux (dynamic combinator)
                                         (dynamic f)
                                         (dynamic struct)
                                         (dynamic def)

where
  aux dcomb df dstruct@(Dyn ra a) ddef = case ra of
    RInt -> cast (app dstruct)
    RFloat -> cast (app dstruct)
    RChar -> cast (app dstruct)
    RUnit -> cast (app dstruct)
    RSum rl rr -> case a of
      Left l -> aux' (Dyn rl l)
      Right r -> aux' (Dyn rr r)
    RProd rp1 rp2 -> case a of
      (p1,p2) -> let d1 = dynamic (aux' (Dyn rp1 p1))
                  d2 = dynamic (aux' (Dyn rp2 p2))
                  in cast (apply (apply dcomb d1) d2)
    RFunc rf1 rf2 -> error "function found"
    RDyn -> let d1 = app dstruct
              d2 = dynamic (aux' a)
              in cast (apply (apply dcomb d1) d2)
    RType term ra' ep -> let d1 = app dstruct

```

```

                                d2 = dynamic (aux' (Dyn ra' (from ep a)))
                                in cast (apply (apply dcomb d1) d2)
RCon s ra' -> aux' (Dyn ra' a)
where
  aux' a = aux dcomb df a ddef
  app dst = apply' ddef df dst

```

Cuando la estructura es un entero, un real, un carácter o un valor de tipo Unit, se le aplica la función utilizando `apply'` y se realiza el `cast` para obtener el valor de retorno a partir del tipo dinámico. En el caso en que la estructura es una suma o un constructor, aplicaremos la estrategia de manera recursiva para el caso que corresponda. En el caso en que la estructura sea una función se retornará un error. En el caso en que la estructura sea un producto, se combinará el valor de retorno obtenido de aplicarle el paso recursivo al primer valor del producto con el valor de retorno obtenido de aplicarle el paso recursivo al segundo valor del producto. La combinación se hace con el operador de combinación que obtiene `everything` como parámetro. En el caso en que la estructura es de tipo `Dynamic`, se le aplica la función recibida como parámetro a la estructura de tipo `Dynamic` y luego se le aplica la estrategia de manera recursiva a la estructura de tipo `Dynamic`. Los dos valores obtenidos se combinan, quedando de esta forma una estrategia top-down. De forma similar se trabaja cuando la estructura es de tipo definido por el usuario. En este caso se le aplica la función a la estructura primero y luego se le aplica de forma recursiva la estrategia a la forma genérica de la estructura y se combinan los dos resultados de manera que la estrategia seguida sea top-down.

Como se puede apreciar, la segunda alternativa es más general. Dada la función `everything`, la definición de la función que construye la lista de enteros contenidos en una estructura es inmediata.

Sin embargo, podríamos desear que durante la recorrida de la estructura para buscar números enteros, al encontrar un número real se tomara la parte entera de dicho número y se agregara a la lista. Esto no se puede conseguir con la segunda alternativa, y es muy sencillo de realizar con la primera.

Al invocar la función

```

putStrLn $ show $ getInts
                (BTreeNode (Leaf (1::Int)) (BTreeNode (Leaf 2) (Leaf 3)))

```

obtendremos la siguiente salida: `[1,2,3]`

### **4.3.Cambio de valores en una estructura**

Veamos ahora un ejemplo propuesto en [7]. Supongamos la siguiente estructura que modela la integración de una compañía:

```

data Company = C [Dept]
data Dept = D Name Manager [SubUnit]
data SubUnit = PU Employee | DU Dept
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Float
type Manager = Employee
type Name = String
type Address = String

```

Como se puede apreciar, una compañía tiene una lista de departamentos. Cada departamento tiene un nombre, un director y una lista de sub-unidades. Cada sub-unidad puede estar compuesta de un empleado o de un departamento. Los datos de un empleado incluyen sus datos personales y su salario. Los datos personales de un empleado son su nombre y su dirección. El salario contiene un número real para indicar el salario que percibe el empleado. A su vez, un director es un empleado, y el nombre y dirección de un empleado son Strings.

Supongamos que deseamos aumentarles el salario en un porcentaje a todos los empleados de una compañía. El código para conseguirlo es el siguiente:

```
increase :: Float -> Company -> Company
increase k (C ds) = C (map (incD k) ds)
incD :: Float -> Dept -> Dept
incD k (D nm mgr us) = D nm (incE k mgr) (map (incU k) us)
incU :: Float -> SubUnit -> SubUnit
incU k (PU e) = PU (incE k e)
incU k (DU d) = DU (incD k d)
incE :: Float -> Employee -> Employee
incE k (E p s) = E p (incS k s)
incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))
```

Como se puede apreciar, la mayoría de las funciones solo tienen la finalidad de recorrer la estructura hasta encontrar una estructura de tipo `Salary`. A este tipo de funciones se le suele llamar código "boilerplate". A continuación mostramos como construir una función politépica que se encargue de realizar esta misma tarea. Primero veamos la definición de una función que mantiene la estructura sin cambios. El código para conseguirlo será el siguiente:

```
increase f c = cast(aux (dynamic c) f)
where
  aux :: Dynamic -> Float -> Dynamic
  aux d@(Dyn ra a) f = case ra of
    RInt -> d
    RFloat -> d
    RChar -> d
    RUnit -> d
    RSum rl rr -> case a of
      Left l -> Dyn ra (Left $ rCast rl (aux (Dyn rl l) f))
      Right r -> Dyn ra (Right $ rCast rr (aux (Dyn rr r) f))
    RProd rp1 rp2 -> case a of
      (p1,p2) -> Dyn
        ra
        (rCast rp1 (aux (Dyn rp1 p1) f),
         rCast rp2 (aux (Dyn rp2 p2) f))
    RFunc rf1 rf2 -> error "function found"
    RDyn -> Dyn ra (aux a f)
    RType term ra' ep -> Dyn ra
      (to ep
       (rCast
        ra'
        (aux (Dyn ra' (from ep a)) f))
       )
    RCon s ra' -> Dyn ra (rCast ra' (aux (Dyn ra' a) f))
```

Se recorre de manera recursiva la estructura y se retorna la misma estructura sin cambio alguno. Recordemos que la función `rCast :: Rep tau -> Dynamic -> tau` sirve para que en caso de que el valor contenido en el elemento de tipo dinámico sea de tipo `tau`, se retorne dicho valor contenido. Para que nuestra función realice los cambios deseados a los salarios debemos considerar el caso particular en que la estructura de datos sobre la cual estamos trabajando sea de tipo `Salary`. En este caso, aplicaremos el aumento correspondiente. Para ello, agregaremos un caso al pattern matching, antes del caso de `RType`. Veamos como queda:

```
RType (App "Salary" []) ra' ep -> dynamic (incS f (cast d))
```

Podemos apreciar lo sencillo que es reconocer un tipo de datos del tipo deseado y aplicarle la función necesaria. De esta manera tenemos una versión politépica de la función que aumenta los salarios a todos los empleados de una empresa.

Podemos notar que la versión politépica requiere más código que su contraparte no politépica. Sin embargo, supongamos que la estructura fuese más compleja. Por ejemplo, si la compañía en lugar de estar formada por departamentos estuviera dividida en países, que a su vez estuvieran divididos en zonas geográficas, a su vez divididos en provincias, en ciudades, en barrios, y por último en departamentos. En este caso, escribir la versión no politépica sería más engorroso que la versión politépica (que ni siquiera necesitaría ser cambiada para adoptar los nuevos cambios).

Veamos ahora otra alternativa, basada en [7], para implementar esta funcionalidad:

```
increase' :: Float -> Company -> Company
increase' k c = everywhere (incS k) c
```

En este caso contamos con una función `everywhere`, la cual recibe una función y una estructura. La función recibida toma una estructura y retorna una estructura del mismo tipo. La función `everywhere` recorre la estructura recibida aplicando la función donde sea posible. Veamos ahora la implementación de `everywhere`:

```
apply'' :: Dynamic -> Dynamic -> Dynamic
apply'' df@(Dyn (RFunc ra rb) f) da@(Dyn ra' a) =
  if ((uniEq ra ra') && (uniEq rb ra'))
  then (apply df da)
  else da

everywhere :: (Representable a, Representable b) =>
  (a->a)->b->b
everywhere f c = cast (aux (dynamic f) (dynamic c))
  where
    aux :: Dynamic -> Dynamic -> Dynamic
    aux func d@(Dyn rx x) = case rx of
      RInt -> apply'' func d
      RFloat -> apply'' func d
      RChar -> apply'' func d
      RUnit -> apply'' func d
      RSum rl rr -> case x of
        Left l -> apply'' func
          (Dyn rx (Left $ rCast rl (aux func (Dyn rl l))))
        Right r -> apply'' func
          (Dyn rx (Right $ rCast rr (aux func (Dyn rr r))))
      RProd rp1 rp2 -> case x of
```



```

(p1,p2) -> apply''
    func
    (Dyn
    rx
    (rCast rp1 (aux func (Dyn rp1 p1)),
    rCast rp2 (aux func (Dyn rp2 p2))))
RFunc rf1 rf2 -> error "function found"
RDyn -> apply'' func (Dyn rx (aux func x))
RType term ra' ep -> apply''
    func
    (Dyn
    rx
    (to ep
    (rCast
    ra'
    (aux func (Dyn ra' (from ep x))))))
    )
RCon s ra' -> apply'' func
    (Dyn rx (rCast ra' (aux func (Dyn ra' x))))

```

La función `apply''` recibe una función dentro de un elemento de tipo dinámico y una estructura también dentro de un elemento de tipo dinámico. Si la función se puede aplicar a la estructura, entonces `apply''` retorna el valor resultante de aplicarle la función recibida a la estructura recibida dentro de un elemento de tipo dinámico. En caso de no poder aplicarle la función a la estructura, se retorna la estructura recibida dentro de un elemento de tipo dinámico.

La función `everywhere` intenta aplicarle la función recibida utilizando `apply''` a la estructura recibida y a sus sub-estructuras de manera recursiva. Así se aplicará la función recibida a las (sub)estructuras que correspondan y el resto de las (sub)estructuras permanecerá incambiada.

Comparativamente, la versión politépica que utiliza `everywhere` requiere menos código que la otra versión politépica. Sin embargo, la versión que no utiliza `everywhere` puede realizar varias modificaciones en una estructura durante una misma recorrida. Supongamos que aparte de incrementar el sueldo de los empleados quiero cambiar el nombre de todos los empleados que se llamen “Junior” por “Jr”. Es muy sencillo modificar la función politépica que no utiliza `everywhere` para que realice estos dos cambios en la estructura en una sola recorrida. Sin embargo, si utilizo `everywhere` debo aplicar la función `everywhere` dos veces, la primera para cambiar los sueldos y la segunda para cambiar los nombres.

Para continuar con el ejemplo, generemos una estructura.

```

genCom :: Company
genCom = C [D "Research" ralf [PU joost, PU marlow],
            D "Strategy" blair []]

ralf, joost, marlow, blair :: Employee
ralf = E (P "Ralf" "Amsterdam") (S 8000)
joost = E (P "Joost" "Amsterdam") (S 1000)
marlow = E (P "Marlow" "Cambridge") (S 2000)
blair = E (P "Blair" "London") (S 100000)

```

Al invocar `putStrLn $ show (increase 0.1 genCom)` obtendremos

```
C [D "Research" (E (P "Ralf" "Amsterdam") (S 8800.0)) [PU (E (P  
"Joost" "Amsterdam") (S 1100.0)),PU (E (P "Marlow" "Cambridge") (S  
2200.0))],D "Strategy" (E (P "Blair" "London") (S 110000.0)) []]
```

## 5. Conclusiones y trabajo futuro

### 5.1. Conclusiones

Hemos comprobado la viabilidad de una extensión de Haskell con las características deseadas.

Se construyó una biblioteca basada en [3] para utilizar tipos dinámicos en Haskell basada en GADTs y se construyó un generador de declaraciones que permiten utilizar tipos de datos no atómicos dentro de valores dinámicos.

También contamos con una herramienta que dado un módulo escrito en Haskell 98 construye otro módulo, el cual contendrá las declaraciones necesarias para trabajar con los tipos de datos definidos en el módulo original dentro de elementos de tipo dinámico.

Consideramos que estos avances nos acercan a la construcción de un framework que permite utilizar tipos dinámicos como si fuera una característica integrada del lenguaje, cumpliendo así con uno de los objetivos del presente proyecto.

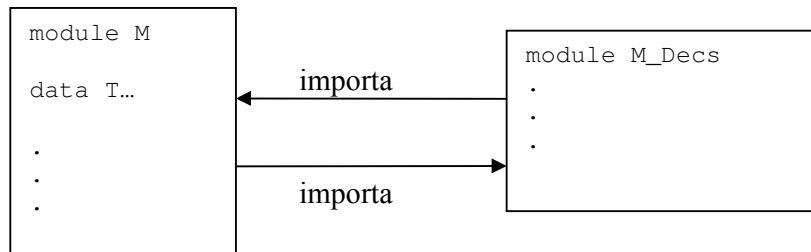
Encontramos las siguientes ventajas de utilizar el enfoque presentado por Hinze y Cheney [3] respecto al abordaje de Cardelli et al. [1]:

- En [1] el lenguaje que se presenta es más simple que el usado en los ejemplos presentados.
- En el enfoque de Hinze y Cheney no es necesario demostrar que ninguna expresión bien tipada lleva a error, como se hace en [1]. Esto se debe a que en nuestro caso no estamos expandiendo el lenguaje, sino que estamos codificando una biblioteca dentro de un lenguaje bien tipado.

### 5.2. Limitaciones

Como limitaciones hemos encontrado:

- Un módulo no puede utilizar tipos de datos declarados en dicho módulo como dinámicos, ya que para ello debe importar el módulo con las declaraciones generadas. Como el módulo de declaraciones generadas importa a su vez el módulo original, se genera una dependencia circular. Veamos un ejemplo.



- No se permiten utilizar extensiones de GHC en el código del usuario. Dicho código debe contener solamente Haskell estándar. Esta limitación surge debido a que para generar los módulos que realizan el reify y el splice de un módulo definido por el usuario se utiliza la biblioteca de parsing de GHC (Language.Haskell.Parser). Actualmente, dicho parser no acepta código que contenga extensiones propias de

GHC, aunque los desarrolladores de GHC afirman que en próximas versiones agregarán dicho soporte.

- Debido a limitaciones de tiempo, se utilizaron los tipos de datos `Either` y pares para representar el producto y la suma de representaciones, así como el tipo `()` de Haskell para representar el propio tipo `Unit`. Al ser usados como tipos básicos en las construcciones de representaciones de tipos dinámicos, ninguno de estos tipos de datos puede ser tratado normalmente dentro de tipos dinámicos. Para utilizar estos tipos de datos normalmente se debería definir en la biblioteca de tipos dinámicos representaciones propias de la suma, el producto y `Unit`, y hacer los cambios correspondientes en el generador de declaraciones.
- No se pueden declarar tipos de datos en un módulo tal que:
  1. Dicho tipo de datos utiliza un tipo de datos definido en otro módulo que no sea el preludio
  2. Se desee generar automáticamente las instancias para que dicho tipo de datos pueda utilizar el constructor inteligente para crear tipos dinámicos partiendo del tipo de datos declarado.

Esta restricción se debe a que, suponiendo que en un módulo `A` se declara un tipo de datos que utiliza otro tipo de datos del módulo `B`, al hacer el splice de las funciones generadas para el módulo `A`, debería necesariamente importar el splice de las funciones generadas para el módulo `B`.

- No se manejan conflictos de nombres. Se debe tener especial cuidado de saber qué nombres se generarán automáticamente y qué nombres y constructores se utilizan en el módulo `Dynamics`. También se deberían manejar los conflictos de tipos de datos con el mismo nombre definidos en diferentes módulos.
- El constructor de tipos dinámicos `RType` no incluye información del módulo en el que está declarado el tipo de datos que representa. Por lo tanto estructuras con el mismo nombre declaradas en módulos distintos se tomarán como iguales aunque no lo son.

### 5.3.Dificultades

En el transcurso del proyecto se encontraron las siguientes dificultades:

- La complejidad del material bibliográfico. No todo el material resultó trivial, especialmente [1] y [9]. Se requirió una buena asimilación de los materiales bibliográficos para comprender el contexto del problema e idear una solución que unifique todos los conceptos.
- La falta de documentación de Template Haskell. Aunque Template Haskell se presenta a grandes rasgos en [11], no están documentadas todas las características de la herramienta, especialmente las que refieren a las estructuras de datos utilizadas. Incluso se han hecho cambios en versiones sucesivas de Template Haskell y la documentación de estos cambios no se encuentra fácilmente.
- El carácter de herramienta experimental de Template Haskell. En particular, durante el desarrollo del proyecto se encontró un error de Template Haskell en la versión 6.4 de GHC el cual se solucionó al pasar a la versión 6.4.1.

### 5.4.Posibles extensiones

Existen extensiones posibles para el trabajo realizado:

- Implementación del framework. En el framework para uso de `Dynamics` se podría automatizar la ejecución de los dos pasos necesarios, siendo estos la

generación del módulo que realiza reify y splice y el compilado de todos los módulos con GHC.

- Soporte para tipos de datos con parámetros de orden mayor a 2. Usando Template Haskell se podrían crear en tiempo de compilación funciones semejantes a *term\*\_\** presentada en [3].
- Soporte para extensiones de GHC en el código del usuario. Si en futuras versiones a la biblioteca de parsing de GHC se agrega soporte para extensiones de GHC, se puede fácilmente implementar la manipulación de dichas extensiones.
- Utilizar estructuras definidas por el usuario para representar el producto y suma de representaciones. En tal caso, para evitar conflictos de nombres, se podría exigir que el módulo Dynamics sea importado como qualified, condición sencilla de comprobar.
- Admitir definiciones de tipos dinámicos que utilizan tipos de datos definidos en módulos diferentes. Para esto, se debería analizar que importaciones se deben incluir al realizar el splice de las funciones generadas para el tipo de datos que utiliza tipos de datos definidos de otros módulos. Más concretamente se deberán importar los splice generados para los tipos de datos definidos en los otros módulos, aunque si dichos tipos de datos se utilizan con qualifiers para evitar conflictos de nombres, podrían existir igualmente conflictos de nombres en las funciones generadas. También se debería incluir en el tipo `RType` información del módulo donde se define el tipo de dato representado. Supongamos que definimos en el módulo A un tipo de datos A y en un módulo B tal que B importa el módulo A definimos un tipo de datos A y otro tipo de datos de la forma `data B = B A.A`. En este caso, al generar las declaraciones para el tipo de datos B, debemos importar el módulo con las declaraciones generadas para el módulo A ya que se utilizará la función de representación del tipo A. Esto requiere un análisis de las importaciones en cada módulo. A su vez, se deben utilizar qualifiers para esta función de representación, ya que el tipo A definido en el módulo B también tendrá una función de representación con el mismo nombre que la función de representación para el tipo A definido en el módulo A. Actualmente no se utilizan qualifiers en las funciones invocadas ni se puede distinguir dentro de los nombres del tipo `RType` el tipo A del módulo A del tipo A del módulo B.
- Manejo de conflictos de nombres. Si se verifica que el módulo Dynamics se importe siempre con la cláusula qualified, se pueden evitar conflictos de nombres definidos en dicho módulo con nombres definidos en otros módulos. También se deberían usar qualifiers al generar las funciones para los distintos tipos de datos y al generar la declaración que define al tipo de dato como instancia de la clase Representable. Si solo se desean importar las declaraciones para declarar a los tipos de datos del usuario como instancias de Representable, sin importar las funciones auxiliares, se puede controlar que al importar los módulos que realizan los splice se importe la lista vacía de funciones. Tal vez estos controles y el agregado de la cláusula qualified al importar el módulo Dynamics podría estar a cargo del framework, resultando más transparente para el usuario.

## 6. Bibliografía

- [1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. *Dynamic Typing in a Statically Typed Language*. In 16th POPL, pages 213-227, 1989.
- [2] Juan José Cabezas. *The C5 Programming Language*. 2005 Edition.
- [3] J. Cheney and R. Hinze, *A lightweight implementation of generics and dynamics*, Haskell Workshop, USA, 2002.
- [4] Ralf Hinze. *Generics for the masses*. In Kathleen Fisher, editor, Proceedings of ICFP'04, Snowbird, Utah, September 19-22, 2004.
- [5] Ralf Hinze. *Fun with phantom types*. In Jeremy Gibbons and Oege de Moor, editors, The Fun of Programming, pp. 245-262. Palgrave Macmillan, 2003.
- [6] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000
- [7] Ralf Lämmel and Simon Peyton Jones. *Scrap your boilerplate: a practical design pattern for generic programming*. ACM SIGPLAN Notices, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
- [8] Konstantin Laufer, *Combining Type Classes and Existential Types*, CLEI, ITESM-CEM, Mexico, September 1994
- [9] Xavier Leroy and Michel Mauny. *Dynamics in ML*. Journal of Functional Programming, 3(4):431-463, 1993.
- [10] Bruno C. d. S. Oliveira and Jeremy Gibbons (2005). *TypeCase: A Design Pattern for Type-Indexed Functions*. Haskell Workshop, September 2005.
- [11] Tim Sheard and Simon Peyton Jones, *Template metaprogramming for Haskell*, Haskell Workshop May 2002.
- [12] [http://www.haskell.org/ghc/docs/latest/html/users\\_guide/gadt.html](http://www.haskell.org/ghc/docs/latest/html/users_guide/gadt.html)