2.1: CAVI-ART Project

2.2: QuickCheck

2.3: Generics

El siguiente punto a tratar en estos preliminares es la librería Generics de Haskell, una librería utilizada principalmente para la generación automática de instancias de funciones correctas para cualquiera que sea el tipo de datos. En el caso de este proyecto Generics apareció como una librería necesaria pues debiamos conseguir escribir funciones como "compose" de manera que funcionaran para cualquier tipo de datos, incluidos los definidos por el usuario y de los cuales no podemos tener conocimiento en adelantado.

La librería Generics dentro de Haskell es posible por dos caracteristicas del propio lenguaje:

1. En primer lugar la existencia de las clases de tipos, que actuan como una interfaz definiendo el comportamiento de los tipos que pertenecen a dicha clase.
2. En segundo lugar, gracias a la existencia del polimorfismo de tipo Ad-hoc. Este nos permite abstraer una operación sobre una o mas clases de tipos simplemente con la condición de que el tipo concreto tenga unas propiedades como por ejemplo que sea ordenable (pertenezca a la clase Ord) o que sus elementos admitan comparaciones por igualdad (que pertenezca a la clase Eq).

En el caso de esta librería, la clase de tipos principal (Generic) expresa la posibilidad de describir un tipo de datos en terminos de un conjunto simple de combinadores. Estos combinadores son:

- En primer lugar debemos definir el comportamiento deseado para los tipos de datos vacios (representados con V1 en Generics)
- En segundo lugar debemos definir el comportamiento deseado para los tipos de datos cuyo constructor carece de parámetros (representados con U1 en Generics).
- En tercer lugar se trata de definir el comportamiento para los tipos compuestos de acuerdo a como se forman. En Haskell los tipos compuestos solo pueden definirse mediante dos operaciones partiendo de los tipos básicos. Estas dos operaciones son la suma y el producto de tipos (representados como :+: y :*: respectivamente en Generics). Deberemos definir como queremos que sea el comportamiento de las funciones de nuestra clase genérica de acuerdo a como se forma a partir de los tipos básicos.
- Por último están dos tipos para representar meta-información y etiquetado de tipos(representados respectivamente por M1 y K1), que nos permitirán

definir el comportamiento esperado para las funciones cuando esta depende de las etiquetas o parte de la meta-información del tipo.

Una vez definidos estos cinco diferentes combinadores es necesario definir algunas instancias para los tipos predefinidos como Int, Char, Boolean... de manera que si el usuario crea un tipo complejo como por ejemplo un Diccionario con valores de Int como clave y Char como valores tengamos un punto de partida para construir mediante Generics las instancias en nuestra clase para los nuevos tipos de datos.

Adjunto codigo en el cual se puede ver un ejemplo de como escribir usando Generics una clase que nos permita serializar cualquier tipo de datos.

En primer lugar definimos un tipo de datos Bit que será la representación de nuestra serialización. Después creamos la clase Serialize que será la interfaz externa y visible. Ademas debemos definir la clase GSerialize que será la clase en la que definiremos usando Generics los comportamientos deseados según sea la forma de nuestro tipo de datos. Por último dentro de Serialize debemos incluir una cláusula default que es la que enlaza la función put en Serialize con la función gput en GSerialize.

```
data Bit = O | I

class Serialize a where
  put :: a -> [Bit]
  default put :: (Generic a, GSerialize (Rep a)) => a -> [Bit]
  put a = gput (from a)

class GSerialize f where
  gput :: f a -> [Bit]
```

Despues definimos las diferentes instancias de la clase GSerialize para los 5 combinadores nombrados con anterioridad.

```
instance GSerialize U1 where
  gput U1 = []

instance (GSerialize a, GSerialize b) => GSerialize (a :*: b) where
  gput (a :*: b) = gput a ++ gput b

instance (GSerialize a, GSerialize b) => GSerialize (a :+: b) where
  gput (L1 x) = O : gput x
  gput (R1 x) = I : gput x

instance (GSerialize a) => GSerialize (M1 i c a) where
  gput (M1 x) = gput x
```

```haskell
instance (Serialize a) => GSerialize (K1 i a) where
 gput (K1 x) = put x
```

Por último debemos definir algunas instancias para algunos tipos predefinidos básicos.

```haskell
instance Serialize Int where
  put i = serializeInt i

instance Serialize Bool where
  put True = [I]
  put False = [O]

instance Serialize a => Serialize [a] where
  put []    = []
  put (h:t) = put h ++ put t
```

2.4 Template Haskell

En este apartado trataremos sobre Template Haskell, una extensión sobre el lenguaje original que añade la posibilidad de realizar metaprogramación en Haskell, de una manera similar al sistema de templates de C++, de ahí su nombre, permitiendo a los programadores computar algunas partes de su programa en tiempo de compilación dependiendo de las necesidades.

Un ejemplo de la idea básica

Imaginad escribir una funcion para imprimir un valor en Haskell siguiendo el estilo de C. Nos gustaría poder escribir algo como esto en Haskell:

```haskell
printf "Error: %s on line %d." msg line
```

El caso es que en Haskell uno no puede definir printf de una manera tan sencilla pues el tipo de printf depende del valor de su primer argumento. En Template Haskell en cambio podemos definir printf de manera que sea definido por el usuario, eficiente y garantice la seguridad de tipos de Haskell.

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

El símbolo $ indica “evaluar en tiempo de compilación”. La llamada a la función printf devuelve a Haskell una expresión que es insertada en el lugar de la llamada después de lo cual se puede realizar la compilación de la expresión. Por ejemplo el codigo entre parentesis:

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

lo traduce en la siguiente expresión lambda en Haskell

```haskell
(\ s0 -> \ n1 -> "Error: " ++ s0 ++ " on line " ++ show n1)
```

Sobre la cual se aplicará la comprobación de tipos y se aplicará sobre *msg* y *line*

Como usar template Haskell

Lo primero que hay que nombrar es el hecho que las funciones de Template Haskell que son ejecutadas en tiempo de compilación estan escritas en el mismo lenguaje que las funciones utilizadas en tiempo de ejecución. Una gran ventaja de esta aproximación es que todas las librerías existentes y habilidades usadas en Haskell pueden ser utilizadas directamente en Template Haskell. Por otro lado, una de las posibles desventajas de esta aproximación puede ser la necesidad de tener que utilizar notaciones como "$" o "[||]" (conocidas como splicing y quasi-quotes) para especificar que partes del codigo se deben ejecutar en tiempo de ejecución y cuales en tiempo de compilación.

En los ejemplos más sencillos como el anteriormente presentado sobre como escribir una función printf en Template Haskell la notación del *splicing* o la *quasi-quotation* pueden resultar de gran ayuda. El problema es que tan pronto como empezamos a hacer cosas más complejas en meta-programación esta notación deja de ser suficiente. Por ejemplo no es posible definir una función para seleccionar el i-ésimo elemento de una tupla de n elementos usando solo esas dos notaciones. Dicha función en Template Haskell sería asi.

```
sel:: Int -> Int -> Expr
sel i n = lam [pvar "x" (caseE (var "x") [alt]])
    where alt :: Match
          alt = simpleM pat rhs

          pat :: Patt
          pat = ptup (map pvar as)

          rhs :: Expr
          rhs = var (as !! (i-1))

          as :: [String]
          as = ["a" ++ show i | i <- [1..n]]
```

Cosificación (Reification)

La *cosificación* es la herramienta presente en Template Haskell para permitir al programador preguntar sobre el estado de la tabla de símbolos que guarda el compilador. Por ejemplo se puede escribir un código como el siguiente:

```
module M where
data T a = Tip a | Fork (T a) (T a)

  repT :: Decl
  repT = reifyDecl T
```

```
lengthType :: Type
lengthType = reifyType length

percentFixity :: Q Int
percentFixity = reifyFixity (%)

here :: Q String
here = reifyLocn
```

La primera de las funciones declaradas devuelve un resultado de tipo **Decl** (equivalente a Q Dec), representando la declaración del tipo T. El siguiente computo **reifyType length** devuelve un resultado de tipo Type (equivalente a Q Typ) representando el conocimiento del compilador sobre el tipo de la función **length**. En tercer lugar reifyFixity devuelve la ¨fixity¨ de los argumentos de la función lo cual es muy util cuando se quiere deducir como imprimir algo. Finalmente reifyLocn devuelve un resultado de tipo *Q String* que representa la posición en el código fuente desde donde se ejecutó reifyLocn.

De esta manera la cosificacion devuelve un resultado que puede ser analizado y utilizado en otros cálculos, pero hay que recordar que al tratarse de una herramienta del lenguaje para acceder a la tabla de símbolos y estar encapsulado dentro de la monada Q no puedes usarla como una funcion no se puede usar por ejemplo con la función map (map reifyType xs sería incorrrecto).

Esta parte en concreto de Template Haskell nos resultó de gran ayuda en nuestro proyecto, pues nos permite poder investigar los tipos definidos por el usuario para,dependiendo de cual sea la estructura interna de los mismos poder crear un generador de casos de prueba que tenga sentido y un correcto funcionamiento.