TFG

2. Preliminares

2.1: CAVI-ART Project

The first point I would like to talk about in this preliminary section is the CAVI-ART project, that consists of a project being developped here at the UCM of which my Final Degree Project is part.

The **CAVI-ART** platform consists of a set of tools aimed at assisting the programmer in the validation of programs written in a variety of languages. These aids include the automatic extraction and proving of verification conditions, the automatic proving of termination (whenever this is decidable by using the state-of-the-art technology), the automatic inference of some invariants, and the automatic generation and running of test-cases. Some of its results have been reported elsewhere.

A key aspect of the platform is its Intermediate Representation of programs (in what follows, IR). Source programs written in conventional languages such as C++, Java, Haskell, OCaml, and others, are translated to the IR, so that all the above mentioned activities are performed at the IR level. The intention is to program most of the platform tools once forever, independently of the source language at hand.

The design of the IR was done with the aim of easing the above tasks as much as possible. Our experience on static analysis led us to avoid the complications of mutable state, and also to a simple design having very few primitive constructions. The result was an IR very near to a desugared functional language with flattened expressions. We never thought on the IR as being executable code, but rather we looked at it as an abstract syntax on which it were easy to perform static analysis and formal verification .But on the last months we decided to convert the IR into executable code to make it possible running language independent tests, and also building language independent testing tools. Most of the work done on testing, and almost all the existing testing tools are tied to a particular language.

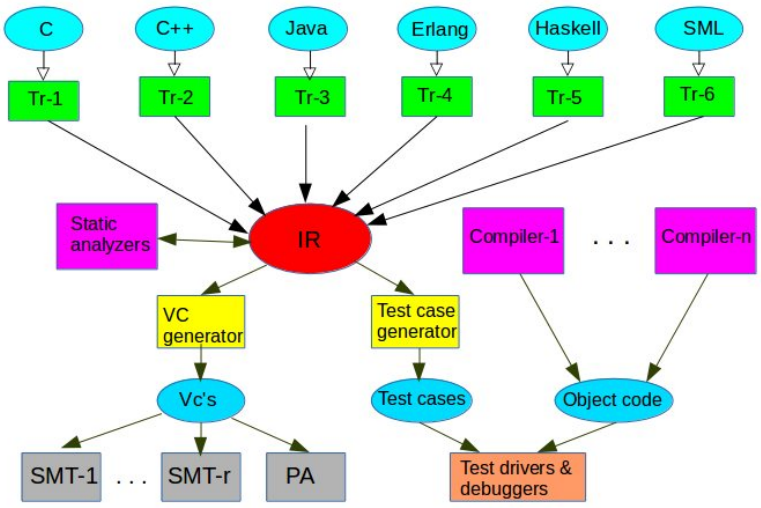

Figure 1: Scheme of CAVI_ART project

2.2: QuickCheck

El segundo punto de los preliminares está dedicado a **Quickcheck** (Claessen and Hughes 2000) una herramienta de Haskell pensada para testear funciones escritas en dicho lenguaje sobre un conjunto de casos de prueba generados de manera aleatoria. Dicho programa resultó ser de gran ayuda pues tiene ideas similares a lo que queríamos conseguir con nuestro proyecto ya que se trata tambien de un sistema de caja negra. Aunque también cuenta con algunas diferencias sobre todo en la generación de los casos de prueba, ya que **Quickcheck** los genera de manera aleatoria mientras que nuestro proyecto los genera de manera exhaustiva.

Ejemplo de funcionamiento del programa

En primer lugar vamos a observar un ejemplo de una función **reverse** que se encarga de invertir una lista, la cual cumple las tres siguientes reglas que son ciertas para cualquier lista finita:

```
reverse [x] = [x]
reverse (xs++ys) = reverse ys++reverse xs
reverse (reverse xs) = xs
```

Vamos a escribirlas ahora como funciones de Haskell para ser asi capaces de

testearlas:

```
prop_RevUnit x =
  reverse [x] == x

prop_RevApp xs ys =
  reverse (xs++ys) == reverse ys++reverse xs

prop_RevRev xs =
  reverse (reverse xs) == xs
```

Ahora lanzamos el programa **Quickcheck** para comprobar si pasa todos los casos de prueba.

```
Main> quickCheck propRevApp
OK: passed 100 tests.
```

El programa nos comunica que de los 100 casos de prueba (que es la cantidad por defecto pero puede cambiarse) nuestro programa pasa todos ellos. Veamos ahora que pasa en caso de que nuestra función no esté bien definida. Definiremos para ellos una de las anteriores de manera incorrecta.

```
prop_RevApp2 xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

y llamamos a dicha función desde **quickcheck**

```
Main> quickcheck prop_RevApp2
Falsifiable, after 1 tests:
[2]
[-2,1]
```

Aquí podemos observar que en caso de fallo **quickcheck** nos devuelve el contraejemplo a nuestra función. Hay que tener en cuenta que siempre nos devolverá un contra ejemplo de tamaño mínimo. Lo que nos dice esta vez esque nuestra definición ha fallado en el primer test y que en dicho caso las respectivas listas para las que ha sido probado falso son [2] y [-2,1].

Leyes condicionales

En algunos casos las leyes que queremos definir no pueden ser representadas mediante una simple función y solo son ciertas bajo unas precondiciones muy concretas. Para dichos casos **Quickcheck** cuenta con el operador de implicación ==> para representar dichas leyes condicionales. Por ejemplo una ley tan simple como la siguiente:

```
x <= y ==> max x y == y
```

Puede ser representada por la siguiente definición.

```
prop_MaxLe :: Int -> Int -> Property
prop_MaxLe x y = x <= y ==> max x y == y
```

O la propiedad de inserción en listas ordenadas.

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==> ordered (insert x xs)
```

En ambos ejemplos podemos observar que el resultado de la función es de tipo *Property* en vez de *Bool*, lo cual es debido a que en el caso de las leyes condicionales en vez de probar la propiedad por para 100 casos de prueba aleatorios, esta es probada contra 100 casos que cumplan la precondición establecida. Si uno de los candidatos no cumple la propiedad este será descartado y se considerará el siguiente. Debemos tener en cuenta una cosa más en cuanto a las leyes condicionales para aquellas en las que la condición rara vez sea satisfecha podemos llegar a un mensaje como este.

```
Arguments exhausted after 64 tests
```

Lo cual significa que despues de generar el máximo número de casos de prueba (que por defecto son 1000) solo ha encontrado 64 de ellos que cumplan la condición. Dicho límite esta pensado para que el programa no busque indefinidamente en caso de que no haya más casos que cumplan la precondición si no que lo intente con un número razonable de casos de prueba, si no encuentra 100 tests válidos antes de dicho número entonces el programa simplemente anuncia cuantos tests pudo realizar correctamente.

Monitorizando los datos

Al testear propiedades debemos tener cuidado, pues quizás parezca que hemos probado una propiedad a fondo para estar seguros de su credibilidad pero esta simplemente ser aparente. Voy a intentar ejemplificarlo añadiendo unos cambios a la función anterior **prop_Insert**.

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    classify (null xs) "trivial" $
      ordered (insert x xs)
```

Lo cual nos permite monitorizar cuantos de los casos que satisfacen la condicion consisten en una inserción sobre una lista vacia, en cuyo caso la condición de **ordered xs** es trivial. Si ejecutamos esta nueva función con **Quickcheck** obtenemos el siguiente mensaje.

```
Ok, passed 100 tests (43% trivial)
```

Es decir que el 43% de los tests realizados son sobre una lista vacia.

Pero a su vez **quickcheck** nos ofrece la posibilidad de un mejor análisis, más alla de etiquetar uno de los casos que nos interese. Podemos realizar una especie de histograma, utilizando la palabra reservada **collect**, que nos dará una mayor información de la distribución de los casos de prueba, por ejemplo en este caso según su longitud.

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    collect (length xs) $
      ordered (insert x xs)
```

Al ejecutarlo obtendriamos un resultado como el siguiente.

```
Ok, pased 100 tests.
49% 0.
32% 1.
12% 2.
4% 3.
2% 4.
1% 5.
```

Lo cual nos permite observar que de los 100 casos de prueba que cumplían la condición solo 19 de ellos trabajan con listas mayores de tamaño 1, lo cual es uno de los grandes problemas de los generadores por defecto que nos proporciona **quickcheck** y para poder solucionar dicho problema se nos proporciona la posibilidad de definir nuestros propios generadores.

Definir generadores

En primer lugar vamos a empezar definiendo la clase de tipos **Arbitrary** de la cual un tipo es una instancia si podemos generar casos aleatorios de él. La manera de generar los casos de prueba depende por supuesto del tipo.

```
class Arbitrary a where
  arbitrary :: Gen a
```

Gen es un tipo abstracto representando el generador para el tipo **a**, que bien puede ser el generador por defecto o uno creador por el programador para el caso específico. El tipo abstracto **Gen** se define como:

```
newtype Gen a = Gen (Rand -> a)
```

En esta definición **Rand** se trata de un número semilla aleatorio y un generador no es más que una función que puede generar una **a** de una manera pseudoaleatoria.

Ahora vamos a echarle un vistazo a las posibilidades que nos ofrece **Quickcheck** a la hora de definir los generadores de casos para los tipos de datos definidos por el usuario. El programa se basa en la idea de que a pesar de que se podrían generar dichos generadores mediante un preprocesamiento prefirieron optar por la idea de que sea el usuario el que los defina personalmente para de esta manera

mantener la herramienta lo más ligera posible y a la vez que cuando el usuario quiera probar una propiedad no tenga que esperar a que se realice todo el preprocesamiento si no que la prueba sea lo más rápida posible.

Supongamos que definimos el tipo **Colour** de la siguiente manera

```
data Coulour = Red | Blue | Green
```

Un ejemplo de un generador para dicho tipo en el cual los tres colores son equiprobables sería

```
instance Arbitrary Colour where
  arbitrary = oneof
    [return Red | return Blue | return Green]
```

en el cual podemos observar el funcionamiento de la función **oneof** que se encarga de devolver uno de los elementos de la lista dando la misma probabilidad a todos ellos.

Vamos a observar otro ejemplo, un generador para listas de un tipo **a** arbitrario.

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
    [ (1, return [])
      (4, liftM2 (:) arbitrary arbitrary)]
```

En ella usamos la función **frequency** la cual funciona similar a **oneof** pero dandole pesos diferentes a los diferentes casos. En este ejemplo le damos peso 1 a la lista vacia y peso 4 a la lista compuesta de otras 2 listas, con lo cual obtenemos casos de prueba de una longitud media de 4. Teniendo en cuenta el ejemplo anterior visto sobre la inserción en una lista ordenada en la cual la mayoria de los casos de prueba eran de longitud 0 o 1 podemos observar que es interesante definir los generadores manualmente ya que esto produce unos mejores casos de prueba.

2.3: Generics

El siguiente punto a tratar en estos preliminares es la librería **Generics** (Magalhaes et al. 2010) 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, incluídos los definidos por el usuario y de los cuales no podemos tener conocimiento en adelantado.

Dicha librería 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 más 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 términos 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 nuestro tipo 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 definidas las funciones para 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 más abajo 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

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

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

```
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** (Sheard and Jones 2002), 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 parte de la generación de código en tiempo de compilación dependiendo de las necesidades.

Un ejemplo de la idea básica

Imaginemos que escribir una funcion para imprimir un valor en Haskell siguiendo el estilo de C. Nos gustaría poder escribir algo como esto en 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 su tipo 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.

```
$(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:

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

lo traduce en la siguiente expresión lambda en 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 resaltar es el hecho que las funciones de **Template Haskell** que son ejecutadas en tiempo de compilación están 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 técnicas 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 código 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 -> ExpQ
sel i n = [| \x -> $(caseE [| x |] [alt]) |]
    where alt :: Match
          alt = simpleM pat rhs

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

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

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

Para explicar un poco este código vamos a empezar de abajo a arriba, para entender las partes que usaremos despues en la función principal **sel**. En primer lugar **as** lo que hace es crear una lista de nombres de aes desde a1 hasta an. La segunda de ellas, **rhs** se encarga de coger el i-esimo elemento de la lista de aes y devolverlo como una variable de tipo **ExpQ** que es el tipo utilizado en **Template Haskell** para las expresiones. La función **pat** transforma en primer lugar la lista de **Strings** en una lista de variables de tipo **PatQ** que es el tipo utilizado en TH para referirse a los patrones y después junta dicha lista en una tupla de tipo **PatQ**. Después al realiza un emparejamiento de la tupla tipo **PatQ** con el **rhs** mediante la función **simpleM** (simple Match). Finalmente la función **caseE** que toma como parametros una variable x (de tipo **ExpQ** como indica la quasi-quotation alrededor de x) y el emparejamiento devuelto por **alt**, realizando la sustituición de la **x** al lado izquierdo de la flecha por el patrón correspondiente y colocando al lado izquierdo de la flecha la **ExpQ** devuelta por **rhs** que es elemento de la tupla tomado.

Esta función se traduciria a una expresión lambda que por ejemplo si llamaramos a **sel 4 6** es decir el cuarto elemento de una tupla de 6 tendría esta forma

```
(\(a1,a2,a3,a4,a5,a6) -> a4)
```

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 puede ser usada como una función, por ejemplo con la función map (**map reifyType xs** sería incorrrecto).

3. Las clases Allv, Sized, Arbitrary

3.1: Black box testing en nuestro contexto

En cuanto al mundo del testing exiten dos grandes posibilidades, sistemas de caja negra o sistemas de caja blanca. En primer lugar los de caja negra son aquellos sistemas de testing que no profundizan en la estructura interna si no que trabajan únicamente con la entrada sobre la que aplican una precondición y la salida sobre la que comprueban si cumple las postcondiciones establecidas. En cambio los de caja blanca testean no solo las entradas y salidas del programa aplicandoles precondiciones y comprobando la postcondiciones si no que además profundizan en la estructura interna del programa para realizar algunas pruebas internas sobre por ejemplo estructuras de datos o el metodos auxiliares del mismo.

En el caso de nuestro proyecto nos decidimos por el método de caja negra pues queríamos conseguir un sistema válido para poder probar cualquier programa sin necesidad de tener que adaptar nuestra plataforma para cada nuevo programa, es decir que funcionase fuese cual fuese el programa bajo prueba. Esa es una de las desventajas del testeo de caja blanca que para poder comprobar partes de la estructura interna de un programa tienes que adaptar la plataforma para cada uno de los nuevos programas.

La idea principal detras de nuestro proyecto era principalmente la inmediatez y la comodidad del usuario, es decir que para probar un programa no necesitara escribir código extra aparte del ya existente programa si no que solo debe especificar como quiere que se generen los casos de prueba y los rangos de los dominios a usar y con eso ser ya capaz de probar su programa lo cual se ajusta mucho más a la idea de testeo de caja negra.

Las posibles maneras en las que el usuario puede especificar que se generen los casos de prueba para cada argumento son 3: - O generar n casos de prueba de manera aleatoria. - O coger n casos de prueba de tamaño menor o igual a m. - O coger los n primeros casos de prueba de la lista de todos los valores, sea cual sea su tamaño.

3.2: Sized

En la estructura del proyecto **Sized** está pensada como la clase externa que hereda de **Allv**. A su vez es la clase que se ocupa de a partir de la lista **allv** de un tipo de datos devolver la lista de los casos de prueba. Esto se realiza mediante dos funciones: - **sized** que devuelve los n primeros casos menores o iguales a un tamaño m. - **smallest** que devuelve los n primeros casos de la lista **allv** según su posición y sin importar su tamaño.

```
class (Allv a) => Sized a where
  -- This function returns the first n elements of size lower or equal m from the "allv" l
```

```haskell
sized::Int->Int->[a]
sized n m = take n (filter (\x-> (size x) <= m) allv)

--This function takes an integer n and returns the n first elements of the "allv" list
smallest::Int->[a]
smallest n = take n allv
```

En esta clase del proyecto decidimos implementar el concepto de tamaño de un elemento mediante la librería **Generics** explicada anteriormente pues de esa manera podríamos tener una representación del tamaño independiente del tipo y no hay que definirlo para cada tipo nuevo creado por el usuario.

En primer lugar debemos definir la clase externa de la parte de **Generics** que será la que nosotros usemos. En ella solo debemos definir las funciones que queremos que tenga y como se comunica con la clases internas de **Generics**. Primero definimos la funcion en si que será una lista que dada un elemento de un tipo cualquiera nos devuelva un entero que representará su tamaño.

```haskell
size ::  a -> Int
```

Despues debemos definir como se comunica la función **size** externa con la versión genérica **gsize** para obtener de esta el valor a devolver. En este caso usamos la función **from** que lo que hace es transformar un valor en su representación no genérica y transformarlo a su representación genérica para que pueda ser manipulado en las diferentes funciones. En este caso es simple pues el valor del tamaño obtenido por **gsize** será el mismo devuelto por nuestra función **size**

```haskell
size a = gsize (from a)
```

En primer lugar creamos la clase interna **GSized** y definimos la función **gsize**

‘haskell -- | This is the generic, non-visible class class GSized f where gsize :: f a -> Int

En el caso base, un constructor sin argumentos el tamaño devuelto por **gsize** es 0.

‘haskell -- | Unit: used for constructors without arguments instance GSized U1 where gsize U1 = 0

En cambio cuando se trata de un tipo compuesto por otros dos tipos, el tamaño del tipo es la suma de los tamaños de los tipos que los componen.

```haskell
-- | Products: encode multiple arguments to constructors
instance (GSized a, GSized b) => GSized (a :*: b) where
  gsize (x :*: y) = gsize x + gsize y
```

En esta tercera instancia definimos el comportamiento cuando el tipo tiene mas de un constructor posible. En este caso si elegimos el constructor de la derecha el tamaño del tipo será el tamaño del tipo de la derecha y similar si elegimos el tipo de la izquierda.

```
-- | Sums: encode choice between constructors
instance (GSized a, GSized b) => GSized (a :+: b) where
  gsize (L1 x) = gsize x
  gsize (R1 x) = gsize x
```

Podemos observar que la función sólo está definida para el caso en el que tenemos exactamente dos constructores diferentes pero esto es debido a que el operador asocia de dos en dos. Por ejemplo si tuvieramos un tipo defindo:

```
data MyExp = Const Int  | Prod MyExp MyExp | Var Char | Sum MyExp MyExp
```

se aplicaría primero sobre los constructores **Const** y **Prod**, despues sobre **Var** y **Sum** y finalmente sobre los resultados de cada pareja.

Por último tenemos la instancia utilizada para trabajar con metainformación del tipo. En el caso de **gsize** al no ser necesaria dicha información simplemente llamamos de nuevo a la funcion **gsize** ignorandola.

```
-- | Meta-information (constructor names, etc.)
instance (GSized f) => GSized (M1 i c f) where
  gsize (M1 x) = gsize x
```

3.3: Allv/TemplateAllv

En primer lugar vamos a tratar la clase **Allv**, cuyas instancias cuentan unicamente con una función, **allv** la cual devuelve la lista de todos los posibles valores del tipo de datos.

Al principio esta clase estaba pensada para ser una única clase que utilizara la librería **Generics** y para contar con un método, **compose** con el cual ser capaces de generar instancias de la clase **Allv** para los tipos definidos por el usuario. Dicha función se encargaría de crear la lista de todos los valores (**allv**) para el nuevo tipo de datos a partir de las listas de los tipos predefinidos, pero encontramos un problema a la hora de integrarlo con la clase **Sized**. La idea que teníamos sobre **Sized** era darle al usuario la posibilidad de pedir los n valores mas pequeños de una clase o los n primeros valores de tamaño menor o igual a un número prefijado por él. Lo cual entraba en conflicto con la manera en la que generábamos las listas de **allv** para los tipos definidos por el usuario. Para realizar la composición de dos listas seguimos el método mostrado en la siguiente figura.

//////////IMAGEN EXPLICATIVA COMBINE////////////////////

Dadas dos listas a combinar vamos realizando la combinación de las listas por diagonales, asegurandonos asi de que aunque alguna de las dos listas o ambas sean infinitas no vamos a dejar ningun valor del nuevo tipo de datos sin calcular por el medio, el problema de generar la nueva lista por diagonales viene de que dicha lista no se encuentra ordenada por tamaño de sus elementos. Dicha combinación de listas infinitas podía ser realizada sin problemas usando **Generics** pero el

problema llegaba a la hora de querer devolver los n primeros valores de un tamaño menor o igual a m ya que para ello debíamos ordenar la lista infinita y encontramos el problema de que en dichas listas infinitas los elementos de un tamaño siempre eran infinitos y que siempre habría algun elemento a mayores de tamaño menor o igual a m aunque fuera despues de muchos elementos por el medio que no lo fueran. Dicho problema fue el por que tuvimos que pensar en utilizar **Template Haskell** en lugar de **Generics**.

En la versión definitiva del programa en la clase **TemplateAllv** se encuentra esta funcionalidad de crear una instancia de **Allv** para los tipos de datos definidos por el usuario utilizando para ello **TemplateAllv**, con la ayuda de la ya nombrada funciónn **compose** que tiene la siguiente forma.

```
compose ::[a] -> [b] -> [(a,b)]
compose xs ys = (e:lattice)
  where e:lattice = concat $ diags 0 0 0 xs ys

--
-- It builds the lattice of tuples from two lists, each one may be either
-- finite or infinite


diags :: Int -> [a] -> [b] -> [[(a,b)]]
diags _ [] [] = [[]]
diags i xs ys
    | fullDiag     = [tup k | k <- [0..i]] : diags (i+1) xs ys
    | finiteFirst  = diags (i-1) xs  ysr
    | finiteSecond = diags (i-1) xsr ys
    | otherwise    = diags (i-2) xsr ysr

  where xs'          = drop i xs
        ys'          = drop i ys
        xsr          = tail xs
        ysr          = tail ys
        fullDiag     = not (null xs') && not (null ys')
        finiteFirst  = null xs' && not (null ys')
        finiteSecond = not (null xs') && null ys'
        tup k        = (x,y)
                       where x = xs !! k
                             y = ys !! (i-k)
```

Compose se encarga de concatenar todas las diagonales en una única lista final que es la que se devuelve mediante la función **allv**, por otro lado **diags** se encarga de crear una de las diagonales y mientras no sea la ultima diagonal volver a llamarse a si misma con los parametros para la siguiente. Los parámetros de la función **diags** son: - **i** se trata del ordinal de la diagonal que vamos a generar. - **xs** e **ys** se tratan de las dos listas que vamos a combinar.

La otra parte de **TemplateAllv** que se encarga de investigar el tipo de datos del usuario y adaptar la instancia de **Allv** de dicho tipo se divide en cuatro grandes pasos o funciones.

La primera de ellas **typeInfo** es la que se encarga de extraer la información del tipo declarado por el usuario. Se trata de una función que recibe como parametro una variable del tipo **DecQ** y devuelve una tupla dentro de la monada **Q** con la siguiente información: - En primer lugar el nombre simplificado del tipo del cual vamos a realizar la instancia, refiriendome con simplificado a quitar toda la parte del nombre que se refiere a la estructura de módulos de la cual se hereda dicho tipo. Por ejemplo si crearamos una instacia para el tipo **Integer** del módulo **Prelude** el nombre del tipo sin simplificar sería **Prelude.Integer** y una vez simplificado simplemente **Integer**. Dicho nombre se trata de una variable de tipo **Name**, que es la manera en la que se maneja el tipo **String** en **TH** para todo tipo de nombres. - El segundo se trata del nombre del tipo sin simplificar, devuelto tambien como una variable de tipo **Name**. - El tercero es una lista de enteros para cada uno de los diferentes constructores del tipo. Los enteros expresan el número de argumentos de cada uno de uno de los constructores. - El cuarto se trata de una lista de los diferentes nombres de los constructores de tipo. Se trata de una lista de tipo **Name** - El último se trata de una lista de listas. Cada una de las listas internas contiene los tipos para uno de los constructores del tipo.

En el código a continuación muestro tanto el código para la función **typeInfo** como para la funcion **simpleName** que es la encargada de simplificar el nombre de la función.

```
typeInfo :: DecQ -> Q (Name, Name,[Int],[Name],[[Type]])
typeInfo m =
     do d <- m
        case d of
           d@(DataD _ _ _ _ _) ->
            return $ (simpleName $ name d, name d , consA d, termsA d, listTypesA d)
           d@(NewtypeD _ _ _ _ _) ->
            return $ (simpleName $ name d, name d , consA d, termsA d, listTypesA d)
           _ -> error ("derive: not a data type declaration: " ++ show d)

     where
        consA (DataD _ _ _ cs _)    = map conA cs
        consA (NewtypeD _ _ _ c _)  = [ conA c ]

        conA (NormalC c xs)         = length xs
        conA (RecC c xs)            = length xs
        conA (InfixC _ c _)         = 2

        nameFromTyVar (PlainTV a)    = a
        nameFromTyVar (KindedTV a _) = a
```

```
        termsA (DataD _ _ _ cs _)   = map termA cs
        termsA (NewtypeD _ _ _ c _) = [ termA c ]

        termA (NormalC c xs)      = c
        termA (RecC c xs)         = c
        termA (InfixC t1 c t2)    = c

        name (DataD _ n _ _ _)      = n
        name (NewtypeD _ n _ _ _)   = n
        name d                      = error $ show d

        listTypesA (DataD _ _ _ cs _)    = (map typesA cs)
        listTypesA (NewtypeD _ _ _ c _)  = [ typesA c ]

        typesA (NormalC _ xs)         = map snd xs
        typesA (RecC _ xs)            = map (\(_, _, t) -> t) xs
        typesA (InfixC t1 _ t2)       = [snd t1] ++  [snd t2]

simpleName :: Name -> Name
simpleName nm =
   let s = nameBase nm
   in case dropWhile (/=':') s of
        []          -> mkName s
        _:[]        -> mkName s
        _:t         -> mkName t
```

Las otras tres funciones se encargan de utilizar dicha información extraída del tipo de datos para crear una instancia adecuada de la clase **Allv** para dicho tipo de datos.

La primera de ellas y la más externa en dicho proceso es **gen_allv**, la cual ademas de llamar a **typeInfo** para extraer la información del tipo y pasarsela a las subfunciones también es donde se define como se formará exactamente la nueva función **allv** dentro de la instancia del tipo. Adjunto el código de la función **gen_allv**.

```
gen_allv :: Name -> Q Dec
gen_allv typName =
  do (TyConI d) <- reify typName
      --Extract all the type info of the data type
     (t_name,noSimplifiedName,cInfo,consts,typesCons) <- typeInfo (return d)
     --We call to gen_instance with a name for the class, the name of the constructor,
     --a list of info of the constructors, the constructors itself, lists containing
     --the constructors, name of the data-type without being simplified, and lastly
     --the function to generate the body of the function of the class.
     i_dec <- gen_instance (mkName "Allv") (conT t_name) cInfo consts
```

```
                        typesCons noSimplifiedName (mkName "allv", gen_body)
    return i_dec -- return the instance declaration
           -- gen_body is the function that we pass as an argument to gen_instance
           --and later on is used to generate the body of the allv function
           --for a determined data-type
      where gen_body :: [Int] -> [Name] -> [Name]-> [ExpQ]
            gen_body _ [] [] [] = []
            gen_body (i:is) (c:cs) (f:fs) --cInfo consts listOfF
               | null cs = [appsE (mapE:constructorF:(allvFunc i))] ++
                             (gen_body is cs fs rs)
               | otherwise = [appsE (varE '(++):[appsE (mapE:constructorF:
                           (allvFunc i))] ++ gen_body is cs fs rs)]
                     where --constructorF decided to use the data constructor
                           --if having just one parameter or to use a function if
                           --having more than one. This is duo to the fact that
                           --if the data constructor has more than one parameter
                           --we need to apply compose to them and then apply a
                           --function over the result of compose.
                           constructorF
                               | i > 1 = varE f
                               | otherwise = conE c
                           --mapE, composeE and allvE are three auxiliar function
                           --that serve to get the expresion equivalent to those 3
                           --functions in template haskell
                           mapE = varE 'map
                           composeE = varE 'compose
                           allvE = varE 'allv
                           moveHead (x1:x2:xs) = x2:x1:xs

                           allvFunc 1 = [appsE [allvE]]
                           allvFunc n = [appsE (composeE:[allvE] ++ allvFunc (n-1))]
```

Vamos a echar un vistazo mas de cerca a dicha función **gen_body** dentro de la clausula **where** y al tipo de comprobaciones o analisis sobre el tipo que realiza. En primer lugar nombrar lo que significa cada una de las cuatro listas que recibe como parámetro dicha función: - La primera de ellas contiene los números de parámetros de cada uno de los diferentes constructores. - La segunda contiene los nombres de los diferentes constructores. - La tercera una lista de nombres de **f's** entre $\mathbf{f_1}$ y $\mathbf{f_n}$ siendo n el número de constructores distintos para el tipo de datos. **Gen_body** se encarga de analizar el número de parámetros de cada uno de los constructores ya que si cuenta con un único parámetro, se puede utilizar el nombre del propio constructor sin ningún problema pero en caso de tener más de un parámetro debido a que compose devuelve la lista compuesta como una lista de tuplas, es necesario utilizar una función auxiliar **f** para realizar la aplicación del constructor sobre los elementos de la tupla en lugar de sobre la tupla en si. Aparte de esto **gen_body** se encarga de utilizando **Template**

Haskell conseguir juntar todas las partes que fueron en parte preprocesadas en **gen_clause**

La siguiente función a tratar, **gen_instance** se encarga únicamente de crear una instancia de la clase **Allv** para el nuevo tipo de datos (parámetro **for_type**) y adjuntar a dicha instancia la definición de la función **allv** que se crea en **gen_clause**. Adjunto el código de **gen_instance**

```
--Construct an instance of class class_name for type for_type
--with a corresponding function  to build the method body
gen_instance :: Name -> TypeQ -> [Int] -> [Name]
                  -> [[Type]] -> Name -> Func -> DecQ
gen_instance class_name for_type cInfo consts typesCons typeName_nosimp func =
  instanceD (cxt [])
    (appT (conT class_name) for_type)
    [(func_def func)]
      where func_def (func_name, gen_func)-- extracts func_name and gen_func
                = funD func_name -- method name
                  -- generate function body
                  [gen_clause gen_func cInfo consts typesCons typeName_nosimp]
```

Por último tenemos la función **gen_clause** que se encarga de crear la definición de la función **allv** para el tipo de datos, usando para ello la función **gen_body** que había sido definida anteriormente en **gen_allv**. Además cuenta con una serie de funciones auxiliares que realizan parte del procesamiento: - **listOfFOut** se encarga de crear la lista de nombres de variables entre $\mathbf{f_1}$ y $\mathbf{f_n}$ para aquellos casos en los cuales los constructores tienen más de un parámetro. - **isRec** devuelve una lista de booleanos en la cual cada posición indica si el constructor en dicha posición es recursivo o no. - **reorderL** se encarga de reordenar los constructores (lo cual es equivalente a las listas con la información por cada constructor) de manera que queden en primer lugar aquellos que no son recursivo y al final los que si lo son. Esto es necesario ya que los constructores recursivos harán uso de aquellos que no lo son y por ello los no recursivos deben definirse en primer lugar. - **gen_wheres** que se encargará de definir las clausulas where necesarias para todos aquellos constructores con más de un parámetro que necesiten utilizar una función auxiliar (que son las representadas por las **f's**). - **tupleParam** crea las tuplas de parámetros para cada una de las funciones auxiliares **f**.

Adjunto el código de **gen_clause**

```
  -- Generate the pattern match and function body for a given method and
-- a given data-type. gen_func is the function that generates the function body
gen_clause :: Gen_func -> [Int] -> [Name] -> [[Type]] -> Name -> ClauseQ
gen_clause gen_func cInfo consts typesCons typeName_nosimp =
      (clause []
             --here we execute the gen_function to generate the body of the function
            (normalB $ head (gen_func cInfoOrd constsOrd listOfFOutOrd))
             --this other one generates the where clause of the function
```

```
        (gen_wheres cInfoOrd constsOrd listOfFOutOrd))
  where --listOfFOut generates a fresh list of "Name" for n different f's
        --this f's are used when one of the data types has more than one
        --parameter
        listOfFOut = listOfF (length consts)
        listOfF 0 = []
        listOfF n = (mkName ("f"++ show n)):(listOfF (n-1))
        --isRec checks which of the constructors of the given data-type
        --are recursive and which others are not. It returns a boolean list
        --where true means to be recursive and false to not to be recursive.
        isRec = isRecAux typesCons
        isRecAux [] = []
        isRecAux (x:xs) = (or $ map (==(ConT typeName_nosimp)) x): isRecAux xs
        --ReorderL reorders all this lists so they have all non recursive
        --type constructors first and all recursive ones at the end
        reorderL = auxFirst cInfo consts listOfFOut isRec 0 False
        auxFirst is cs fs rs n foundRec
            |n > ((length rs)-1) = (is, cs, fs, rs)
            |foundRec && (not (rs!!n)) = auxFirst
                                          ((is!!n):(remove n is 0))
                                          ((cs!!n):(remove n cs 0))
                                          ((fs!!n):(remove n fs 0))
                                          ((rs!!n):(remove n rs 0)) 0 False
            |(not foundRec) && (rs!!n) = auxFirst is cs fs rs (n+1) (not foundRec)
            |otherwise = auxFirst is cs fs rs (n+1) foundRec
        --removes position n from the list (x:xs)
        remove n (x:xs) actPos
            |n==actPos = xs
            |otherwise = x:(remove n xs (actPos+1))

        --This four functions serve to take the reordered lists for those 4 lists
        cInfoOrd = (\(x,_,_,_) -> x) reorderL
        constsOrd = (\(_,x,_,_) -> x) reorderL
        listOfFOutOrd = (\(_,_,x,_) -> x) reorderL
        isRecOrd = (\(_,_,_,x) -> x) reorderL


        --gen_wheres is the auxiliar function that generates the where "clause"
        --of the function when necesary.
        gen_wheres [] [] [] = []
        gen_wheres (n:ns) (c:cs) (f:fs) --gen_wheres numParam consts listOfF
            | n > 1 = funD f (bodyFunc listOfVar c):gen_wheres ns cs fs
            | otherwise = gen_wheres ns cs fs
              where listOfVar = listVariab n
                    listVariab 0 = []
                    listVariab n = (mkName ("x"++ show n)):(listVariab (n-1))
```

```
--generates the body for the functions in the where clause when necessary.
bodyFunc listOfVar constructorName = [clause (tupleParam listOfVar)
                                        (normalB (appsE ((conE constructorName):
                                          (map varE listOfVar)))) []]

tupleParam (v:vs) --tupleParam listVars
    | (null vs) = [varP v]
    | otherwise = [tupP ((varP v):tupleParam vs)]
```

3.4: Arbitrary

3.5: Instancias predefinidas

6. Related

6.1: Korat

La primera de las herramientas que vamos a tratar en este apartado se trata de **Korat** (Boyapati, Khurshid, and Marinov 2002), una extensión de Java que sirve para la generación de casos complejos de prueba a partir de unas restricciones dadas.

La idea detras de **Korat** es que dado un predicado en Java y una función **finitialization** en la cual definimos los dominios para cada uno de las clases del input, es decir los valores válidos para cada una de ellas. **Korat** explora el espacio de estados de las posibles soluciones pero generando solo soluciones no-isomorficas entre si, consiguiendo de esta manera una gran poda de las soluciones no interesantes del espacio de búsqueda.

Espacio de estados

Lo primero que hace **Korat** es reservar el espacio necesario para los objetos especificados por ejemplo en el caso de un **BinTree** reservaria espacio para él y para el número de Nodos que queramos. Por ejemplo si queremos un arbol con tres nodos el vector contendría 8 campos: * 2 para el **BinTree** (uno para la raíz y otro para el tamaño) * 2 campos por cada uno de los 3 nodos (hijo izquierdo/hijo derecho)

Cada uno de los posibles candidatos que considere **Korat** a partir de ese momento será una evaluación de esos 8 campos. Por lo tanto el espacio de estados de búsqueda del input consiste en todas las posibles combinaciones de esos campos, donde cada uno de ellos toma valores de su dominio definido en **finitialization**

Proceso de busqueda

Para conseguir explorar de manera sistemática y completa el espacio de estados **Korat** ordena todos los elementos en los dominios de las clases y los dominios de los campos. Dicho orden de cada uno de los dominios de los campos será consistente con el orden del dominio de la clase y todos los valores que pertenezcan al mismo dominio de clase ocurriran de manera consecutiva en el dominio del campo.

Cada uno de los candidatos de la entrada es un vector de índices de su correspondiente dominio del campo. Teniendo en cuenta que el dominio de la clase **Nodo** en el anterior ejemplo cuenta con 3 elementos [N0,N1,N2] a estos debemos añadir null obteniendo asi un dominio de campo [null, N0, N1, N2] que será el dominio de los campos **raiz** e **hijos derecho e izquierdo** de cada uno de los nodos. El dominio del campo **tamaño** será un único entero, 3.

Tras definir los dominios de cada uno de los campos del vector la busqueda comienza con la inicialización de todos los indices del vector a 0. Tras ello para cada posible candidato fijamos los valores de los campos de acuerdo a los valores en el vector y acto seguido invoca a la funcion **repOk** que es donde el usuario ha definido la precondición de la función. Durante dicha ejecución **Korat** monitoriza en que orden son accedidos los campos del vector y construye una lista con los identificadores de los campos, ordenados por la primera vez en que **repOk** los accede.

Cuando **repOk** retorna **Korat** genera el siguiente candidato incrementando el indice del dominio de campo para el campo que se encuentra último en la lista ordenada construida previamente. Si dicho indice es mayor que el tamaño del dominio de su campo este se pone a cero y se incrementa el indice de la posición anterior y así repetidamente.

Al seguir este método para generar el siguiente candidato conseguiremos podar un gran numero de los candidatos que tienen la misma evaluación parcial. Asimismo podemos estar seguros de que dicha poda no deja fuera ninguna estructura de datos válida porque **repOk** no leyó dichos campos y podría haber devuelto falso independientemente de su valor. Gracias a esta poda de una gran parte del espacio de busqueda **Korat** puede explorar espacios de busqueda muy grandes de manera eficiente, dicha eficiencia de la poda depende del método **repOk** por ejemplo si siempre lee todos los datos de entrada antes de terminar la ejecución forzará a Korat a explorar casi todo el espacio de busqueda sin apenas podas.

El algoritmo de busqueda descrito aquí genera las entradas en orden lexicográfico. Además para los casos en los que **repOk** no es determinista este método garantiza que todos los candidatos para los que **repOk** devuelve true son generados, los casos para los que siempre devuelve false nunca son generados y los casos para los que alguna vez se devuelve true y a veces false pueden ser o no generados.

Resultados no-isomorfos

Dos candidatos se definen como isomorfos si las partes de sus grafos alcanzables desde la raíz son isomorfas. En el caso de **repOk** el objeto raíz es el objeto pasado como argumento implícito.

El isomorfismo entre candidatos divide el espacio de estados en particiones isomórficas (debido al ordenamiento lexicográfico introducido por el orden de los valores de los dominios de los campos y la ordenación de los campos realizado por **repOk**). Para cada una de dichas particiones isomomorficas **Korat** genera únicamente el candidato lexicográficamente menor.

Además con el proceso explicado anteriormente para ir generando el siguiente candidato teniendo en cuenta la lista de ordenación de los campos **Korat** se asegura de no generar varios candidatos dentro de la misma partición isomórfica.

6.2: Smallcheck

La segunda herramienta a tratar en este apartado es **Smallcheck** (Runciman, Naylor, and Lindblad 2008) una librería para Haskell usada en el testing basado en propiedades. Esta librería parte de las ideas del **Quickcheck** y perfecciona algunos de los puntos flacos de este.

Generación de valores pequeños

La principal diferencia de **Smallcheck** respecto a **Quickcheck** se trata de la generación de sus casos de prueba. En este caso **Smallcheck** se apoya en la "hipótesis del ámbito pequeño" la cual dice que si un programa no cumple su especificación en alguno de sus casos casi siempre existirá un caso simple en el cual no la cumpla o lo que viene a ser lo mismo, que si un programa no falla en casos pequeños lo normal es que no falle en ninguno de sus casos.

Partiendo de esta idea cambia la generación existente en **Quickcheck**, la cual era aleatoria, por una generación exhaustiva de todos los casos de prueba pequeños ordenados por **profundidad** (que es el nombre usado para el tamaño), dejando a criterio del usuario hasta que profundidad deben considerarse como pequeños. A continuacion les presento como están definidas las profundidades más importantes:

- En el caso de los tipos de datos algebraicos como es usual la profundidad de una construcción de aridad cero es cero mientras que la profundidad de una construcción de aridad positiva es una más que la mayor profundidad de todos sus argumentos.
- En el caso de las tuplas dicha profundidad se define un poco diferente. La profundidad de una tupla de aridad cero es cero pero la profundidad de una tupla de aridad positiva es simplemente la mayor profundidad de entre todas las de sus componentes.
- En el caso de los tipos numéricos la definición de la profundidad es con respecto a una representación imaginaria como una estructura de datos. De esta manera, la profundidad de un entero i será su valor absoluto, ya que se construyó de manera algebraica como $\mathbf{Succ^i\ Zero}$. A su vez, la profundidad de un numero decimal $\mathbf{s\ x\ 2^e}$ es la profundidad del par de enteros (s,e).

Tipos de datos seriales

Smallcheck define una clase **Serial** de tipos que pueden ser enumerados hasta una determinada profundidad. Para todos los tipos de datos del preludio existen instancias predefinidas de la clase **Serial**. Sin embargo definir una nueva instancia de dicha clase para un tipo de datos algebraico es muy fácil, se trata de un

conjunto de combinadores **cons**, genéricos para cualquier combinación de tipos Serial, donde es la aridad del constructor.

Por ejemplo supongamos un tipo de datos en Haskell **Prop** en el cual tenemos una variable, la negación de una variable y el **Or** de dos variables

```
data Prop = Var Name | Not Prop | Or Prop Prop
```

Definir una instancia de la clase **Serial** para dicho tipo de datos sería como sigue. Asumiendo una definición similar para el tipo **Name**.

```
instance Serial Prop where
  series = cons1 Var \/ cons1 Not \/ cons2 Or
```

Una serie simplemente es una función que dado un entero devuelve una lista finita.

```
type Series a = Int -> [a]
```

A su vez el producto y la suma sobre dos series se definen como:

```
(\/) :: Series a -> Series a -> Series a
s1 \/ s2 = \d -> s1 d ++ s2 d

(><) :: Series a -> Series b -> Series (a, b)
s1 >< s2 = \d -> [(x,y) | x <- s1 d, y <- s2 d]
```

Por último los combinadores **cons** están definidos usando $><$ decrementando y comprobando la profundidad correctamente.

```
haskell   cons0 c = \d -> [c]   cons1 c = \d -> [c a | d > 0,
a <- series (d-1)]   cons2 c = \d -> [c a b | d > 0, (a,b) <-
(series >< series) (d-1)]
```

Trabajando con grandes espacios de busqueda.

Usar el esquema general para definir series de valores de prueba muchas veces produce que para alguna profundidad pequeña **d** los **10.000-100.000** casos de prueba son rápidamente comprobados pero para la profundidad **d+1** resulte imposible completar los miles de millones de casos de prueba. Por ello resulta necesario reducir algunas dimensiones del espacio de busqueda de manera que otras de las dimensiones puedan ser comprobadas en mayor profundidad.

El primer punto a tener en cuenta es que a pesar de que los números enteros pueden parecer una elección obvia como valores bases para las pruebas debemos tener en cuenta que los espacios de busqueda para los tipos compuestos (especialmente funcionales) al usar bases numéricas crecen de manera muy rápida. Para muchas propiedades **Bool** puede ser también una elección perfectamente válida con lo que conseguiríamos reducir en gran medida el espacio de busqueda.

Lazy SmallCheck

Se trata de una versión de **Smallcheck** pero que a su vez se aprovecha de la evaluación perezosa de Haskell, la cual permite que una función devuelva un valor aunque esta esté aplicada sobre una entrada definida parcialmente. Esta posibilidad de ver el resultado de una función sobre muchas entradas en una sola ejecución puede resultar de gran ayuda en el testeo basado en propiedades ya que si una función se cumple para una solución parcial, esta se cumplirá para todas las funciones totalmente definidas que partan de dicha definición parcial. En eso se centra el **Lazy Smallcheck**, en evitar generar todas esas funciones totalmente definidas que no aportan nada de información extra sobre la definición parcial. La actual versión de **Lazy Smallcheck** es capaz de testear propiedades de primer orden con o sin cuantificadores universales.

Uno de los problemas de **Lazy Smallcheck** es que en algunos casos deberemos tener en cuenta en que orden ponemos las condiciones de la función que queremos probar pues según sea dicho orden los resultados pueden ser distintos. Por ejemplo vamos a tomar un ejemplo en el cual queremos testear la función **isOrderedSet** la cual aplicada sobre una lista nos devuelve si esta es un conjunto y a su vez está ordenada o no.

```
isOrderedSet s = ordered s && allDiff s

allDiff [] = True
allDiff (x:xs) = x `notElem` xs && allDiff xs

ordered [] = True
ordered [x] = True
ordered (x:y:zs) = x <= y && ordered (y:zs)
```

Queremos a su vez comprobar la propiedad **prop_insertOrderedSet** la cual comprueba si dado un conjunto ordenado despues de insertar un nuevo elemento en el este sigue siendo un conjunto ordenado.

```
prop_insertOrderedSet c s = isOrderedSet s ==> isOrderedSet (insert c s)
```

Al ejecutar **Lazy Smallcheck** sobre esta última propiedad obtenemos los siguientes resultados:

```
Main> depthCheck 7 prop_insertOrderedSet
OK, required 964 tests at depth 7
```

En cambio si cambiamos de orden las dos partes de la conjunción de la función **isOrderedSet** objenemos los siguientes resultados.

```
Main> depthCheck 7 prop_insertOrderedSet
OK, required 20408 tests at depth 7
```

Esta diferencia de veinte veces más tests necesarios es debido a que el operador **&&** evalua primero la parte izquierda y **ordered** es más restrictivo que **allDifferent** por lo cual en el primer ejemplo muchos de los ejemplos no se

llegan a comprobar por no cumplir **ordered** mientras que en el segundo ejemplo son muchos mas los casos de prueba necesarios por ser **allDifferent** menos restrictiva.

Por suerte existe un nuevo operador en **Lazy Smallcheck**, la conjunción parelela ***&***, la cual es falsa si cualquiera de sus operandos lo son y que resulta de gran utilidad tanto para conseguir que el programa necesite menos casos de prueba como para que no sea importante el orden en el que esten escritos las condiciones de la precondición.

References

Boyapati, Chandrasekhar, Sarfraz Khurshid, and Darko Marinov. 2002. "Korat: Automated Testing Based on Java Predicates." July. http://web.eecs.umich.edu/~bchandra/publications/issta02.pdf.

Claessen, Koen, and AJohn Hughes. 2000. "QuickCheck:A Lightweight Tool for Random Testing of Haskell Programs." September. https://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf.

Magalhaes, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. "A Generic Deriving Mechanism for Haskell." September. http://www.dreixel.net/research/pdf/gdmh_nocolor.pdf.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad. 2008. "SmallCheck and Lazy Smallcheck Automatic Exhaustive Testing for Small Values." September. https://pdfs.semanticscholar.org/2460/c9b40ea3c4bbaef53c5f4ad2717154cf15b5.pdf.

Sheard, Tim, and Simon Peyton Jones. 2002. "Template Meta-Programming for Haskell." October. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meta-haskell.pdf.