Implementación de una herramienta de pruebas basadas en asertos para una plataforma de verificación

Autor: **Pedro García Castillo**

Director: **Ricardo Peña Marí**

Facultad de Informática

Universidad Complutense de Madrid

Curso 2016-2017

13 de septiembre de 2017

Índice

Capítulo 1

Resumen

1.1 Resumen

Una de las partes más costosas dentro del desarrollo de programas es el testeo, ya que requiere un gran esfuerzo humano para poder especificar los diferentes casos de prueba, lanzarlos y analizar los resultados. Ello provoca que en la mayoría de los casos los programas se prueben mucho menos a fondo de lo que sería necesario. Por ello, en los últimos años han sido desarrolladas diversas herramientas para automatizar de manera parcial dicho proceso de testeo. Sin embargo la mayoría de ellas están especializadas en un único lenguaje de programación.

Nuestro objetivo es conseguir una plataforma que permita el testeo de aplicaciones de manera automática para el usuario y que admita como entrada un programa escrito en cualquier lenguaje de programación.

En este trabajo vamos a presentar la herramienta **Case Generator**, que se engloba dentro del proyecto CAVI-ART, siendo esta parte la encargada de generar los casos de prueba de manera automatizada, adaptándolos a las necesidades de cada ejecución. Este proyecto toma como base las ideas desarrolladas anteriormente por programas como Quickcheck, Korat o Smallcheck, pero intentando conseguir que el proceso de prueba sea más automático, y a la vez compatible con diversos lenguajes de programación tanto funcionales como no funcionales. Para lograr el primer objetivo hemos eliminado la obligación de que el usuario defina un nuevo generador para cada uno de los nuevos tipos definidos. Así, será el propio programa el que realice la tarea de investigar estos tipos y deducir un generador de casos adecuado para cada uno de ellos. Para lograr el segundo en cambio hemos creado una Representación Intermedia (IR) a la que se traducen los programas antes de ser testeados y que permite escribir una plataforma independiente del lenguaje de programación.

A su vez profundizaremos en la estructura de clases de **CaseGenerator** y explicaremos su código, de manera que queden claras todas las ideas detrás de su funcionamiento y las razones por las que decidimos utilizar algunas tecnologías, como la librería `Generics` del compilador GHC y la extensión de Haskell llamada `Template Haskell`.

Por último, tras explicar el funcionamiento de la herramienta expondremos algunos ejemplos prácticos del funcionamiento del programa al ser ejecutado con funciones reales.

1.2 Summary

One of the most costly parts in software development is testing because it requires a lot of human effort to be able to specify all the test cases launch them and analise all their results. This leads to the problems of most of the programms not being tested as much as it would be necessary. This is the reason why in the last years many testing tools have being developed to automate partialy the testing process. Nevertheless most of them are specialised on a single programming language.

Our objective is building a platform that allows tesing applications automatically for the user and that admits as input a program written in any programming language.

Inside of this project we will talk about the tool called **Case Generator**, that is situated inside the CAVI-ART project, being inside of it the part in charge of generating automatically the test cases, adapting them to the needs of each execution. The project takes some ideas used previously in other programms like Quickcheck, Korat or Smallcheck but pursuing the idea of a much automatic process at the same time that it is compatible with several programming languages (functional and non-functional ones). To do so our first objective is to get rid of the obligation from the user to define a new generator for each of the newly defined datatypes. Doing so it would be the programm itself the one having to analyze those types and to deduct a generator fiting each of them. In order to be able to do this second change we created an Intermediate Representation (IR) to which all programms are translated before being tested which makes posible to write a plattform independent of all programming languages.

In this project we will also explain the class structure of **CaseGenerator** and it's code to make clear all the ideas behind it's behaviour together with why we decided to use some technologies as the library `Generics` of the GHC compiler and the Haskell extension called `Template Haskell`.

Finally after explaining how the plattform works we will show some examples about the program behaviour while executed with real functions.

1.3 Palabras clave

prueba, verificación, automática, caja negra, pruebas basadas en asertos

1.4 Keywords

testing, verification, automatic, black box, assertion based testing

Capítulo 2

Preliminares

2.1 Proyecto CAVI-ART

En esta sección explicamos el proyecto CAVI-ART, actualmente en fase de desarollo en la UCM y del cual forma parte mi TFG.

La plataforma **CAVI-ART** (vease el esquema de la figura 2.1) consiste en un conjunto de herramientas pensadas para ayudar al programador en la validación de programas escritos en diferentes lenguajes. Estas ayudas incluyen la extracción automática y prueba de condiciones de verificación, la prueba automática de terminación (siempre que sea decidible usando la tecnología actual), la inferencia automática de algunos invariantes y la generación automática y ejecución de casos de prueba. [6, 3, 5]

Un aspecto clave de la plataforma es su Representación Intermedia de los programas (de aquí en adelante IR). Los programas escritos en lenguajes convencionales como C++, Java, Haskell, OCaml y otros, se traducen a la IR, sobre la que se realizan todas las actividades mencionadas anteriormente. La intención es programar la mayor parte de la plataforma una sola vez, de manera que sea independiente del lenguaje de programación utilizado.

La IR se diseñó con la intención de facilitar al máximo posible las tareas nombradas con anterioridad, mediante un diseño simple que cuenta con muy pocas construcciones primitivas. Nunca se pensó en la IR como código ejecutable sino como una sintaxis abstracta para facilitar el análisis estático y la verificación formal. Sin embargo en los últimos meses se decidió convertir la IR en código ejecutable, para posibilitar la ejecución de pruebas y construcción de herramientas de testeo, ambas independientes del lenguaje. Esto supone una ventaja ya que la mayoría de las herramientas de testeo existentes están ligadas a un lenguaje en concreto.

La parte del proyecto encargada de traducir la IR a Haskell y hacer ejecutables los asertos se ha realizado dentro del trabajo de fin de grado de Marta Aracil Muñoz con título *Implementación de asertos ejecutables para una plataforma de verificación* que también se engloba dentro del proyecto CAVI-ART.

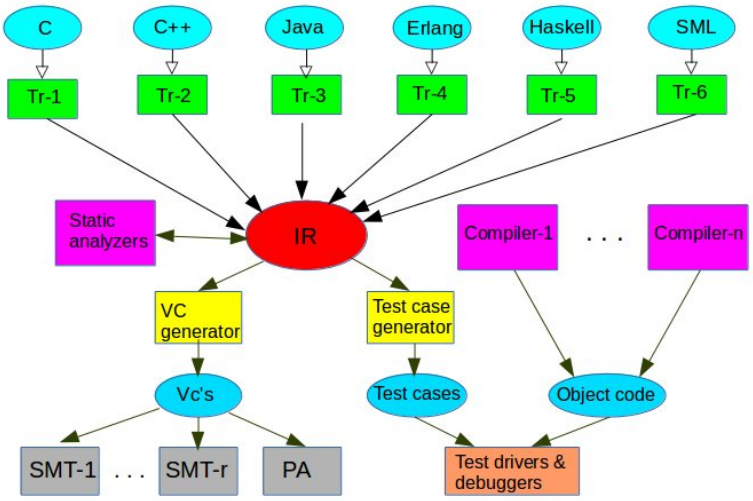

Figura 2.1: Esquema del proyecto CAVI-ART

2.2 QuickCheck

Quickcheck [2] es una herramienta de Haskell pensada para probar 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 también de un sistema de prueba tipo caja negra. Sin embargo presenta 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 generará, como veremos, de manera exhaustiva.

2.2.1 Ejemplo de funcionamiento del programa

En este caso vamos a trabajar con la siguiente propiedad de las listas, cierta para cualquier lista finita.

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

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

```
Main> QuickCheck prop_RevApp
OK: passed 100 tests.
```

Veamos ahora que pasa en caso de que nuestra función no esté definida correctamente.

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

Al ejecutar la nueva funcion 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 de tamaño mínimo, lo que nos indica esta vez es que 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].

2.2.2 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:

$$x <= y \implies \max\ x\ y == y$$

Puede ser representada mediante la siguiente definición.

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

En este ejemplo podemos observar que el resultado de la función es de tipo `Property` en vez de `Boolean`, lo cual es debido a que en el caso de las leyes condicionales en vez de probar la propiedad para 100 casos aleatorios, ésta es probada contra 100 casos que cumplan la precondición establecida. Si uno de los candidatos no la cumple será descartado y se considerará el siguiente. Quickcheck genera un máximo de 1000 casos de prueba y si entre ellos no se han encontrado al menos 100 que cumplan la precondición, simplemente informa al usuario cuantos la cumplen. Dicho límite está pensado para que en caso de que no existan más casos que cumplan dicha precondición el programa no busque indefinidamente.

2.2.3 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 sea solo aparente. Intentaremos ejemplificarlo usando la inserción en una lista ordenada.

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

Esto nos permite conocer cuantas de las pruebas se realizaron sobre una lista vacia. En cuyo caso la condición de `ordered xs` es trivial. Si ejecutamos esta nueva función con Quickcheck obtendremos 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 allá 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, separado según los tamaños de las listas.

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

2.2.4 Como definir generadores

En primer lugar Quickcheck empieza definiendo la clase `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 creado por el programador para el caso específico. El tipo abstracto `Gen` se define como:

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

En esta definición `Rand` es un número semilla aleatorio y un generador no es más que una función que puede crear un valor de tipo `a` de una manera pseudoaleatoria.

Ahora vamos a analizar las posibilidades que nos ofrece Quickcheck a la hora de definir los generadores de casos para los tipos de datos definidos por el usuario.

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, en este caso 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 de manera similar a `oneof`, pero dándole pesos distintos 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 obtendremos 4 veces más casos de prueba de longitud 4 que listas vacias y de esta manera evitaremos el problema indicado anteriormente en el que la mayoría de los casos de prueba eran listas vacias.

2.3 Librería Generics de GHC

El siguiente punto a tratar en estos preliminares es la librería `Generics` del compilador GHC de Haskell [4], una librería utilizada principalmente para la generación automática de instancias de clases de tipos correctas para cualquiera que sea el tipo de datos. En el caso de este proyecto `Generics` apareció como una librería necesaria

para escribir nuestro programa de manera que funcionara para cualquier tipo de datos, incluídos los definidos por el usuario y de los cuales no podemos tener conocimiento por adelantado.

Dicha librería dentro de Haskell facilita la definición de instancias de clase. Las clases y sus instancias son dos características del propio lenguaje:

1. Las clases de tipos, actúan como una interfaz de Java definiendo el comportamiento de las operaciones sobre los tipos que pertenecen a dicha clase.

2. Las instancias permiten el llamado *polimorfismo de tipo Ad-hoc*. Este nos permite sobrecargar funciones tales como el `<=`, el `==` y otras muchas definiendo instancias diferentes para diferentes tipos.

En el caso de la librería `Generics`, ésta permite definir instancias genéricas para cualquiera que sea el tipo al que se apliquen ya que dicha definición se realiza sobre la estructura del tipo y teniendo en cuenta que todo tipo algebraico en Haskell utiliza un número pequeño de construcciones (uniones, productos cartesianos, recursión y tipos básicos).

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 unión disjunta y el producto de tipos (representados como `:+:` y `:*:` respectivamente en `Generics`). Deberemos establecer 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 variables de tipo `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.

2.4 Template Haskell

En este apartado trataremos sobre `Template Haskell` [8], 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.

2.4.1 Un ejemplo de la idea básica

Imaginemos que escribimos una función 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 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 código entre paréntesis:

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

lo traduce a la siguiente expresión lambda en Haskell

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

Sobre la cual se realizará la comprobación de tipos y después se aplicará sobre `msg` y `line`

2.4.2 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 las técnicas usadas en Haskell pueden ser utilizadas directamente en `Template Haskell`. Por otro lado, una de las posibles desventajas de esta aproximación es la necesidad de tener que utilizar notaciones como `$` o `[||]` (conocidas como *splicing* y *quasi-quotes* respectivamente) la primera de ellas traduce las expresiones en `Template Haskell` a expresiones Haskell y la otra realiza la traducción inversa.

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, con el fin de entender las partes que usaremos despues en la función principal `sel`. En primer lugar el cometido de `as` 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 `String` en una lista de variables de tipo `PatQ` que es el utilizado en `TH` para referirse a los patrones y después junta dicha lista en una tupla de tipo `PatQ`. Después realiza un emparejamiento de la tupla tipo `PatQ` con el `rhs` mediante la función `simpleM` (simple Match). Finalmente, la función `caseE` que toma como parámetros 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 el elemento tomado de la tupla.

Esta función se traduciria a una expresión lambda que realizando la llamada `sel 4 6` es decir seleccionar el cuarto elemento de una tupla de 6 tendría esta forma

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

2.4.3 Reification (Cosificación)

La *reification* es una facilidad de `Template Haskell` que permite 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 cómputo `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 resulta muy útil 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 *reification* 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 mónada `Q`, no puede ser usada como una función, por ejemplo con la función map (`map reifyType xs` sería incorrrecto).

Capítulo 3

Nuestra propuesta: las clases Allv, Sized y Arbitrary

3.1 Black box testing en nuestro contexto

En el mundo del testing existen dos grandes posibilidades: sistemas de tipo caja negra y sistemas de tipo caja blanca. Los de caja negra son aquellos sistemas de testing que no se basan 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 no testean únicamente las entradas y salidas del programa aplicandoles precondiciones y comprobando la postcondiciones, sino que además se basan en la estructura interna del programa para realizar la generación de casos de prueba, de forma que se cubra todo el texto del programa. Según el criterio de cobertura deseado se pueden generar casos para ejercitar todas las condiciones o todas las ramas o todos los caminos.

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 volver a generar los casos de prueba cuando cambia la estructura interna del programa. Esa es una de las desventajas del testeo de tipo caja blanca, que para poder comprobar partes de la estructura interna de un programa tendríamos que adaptar la plataforma para cada uno de los nuevos programas.

La idea principal detrás de nuestro proyecto era principalmente la inmediatez y la comodidad del usuario, es decir que para probar un programa no fuera necesario escribir código extra, aparte del ya existente programa, sino que solo fuera especificar como quiere que se generen los casos de prueba y los rangos de los dominios a usar y con eso sea 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 como se generan los casos de prueba para cada argumento son 3:

- Generar n casos de prueba de manera aleatoria.

- Coger n casos de prueba de tamaño menor o igual a m.

- 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` (la cual se puede ver en la figura 3.1). A su vez es la clase que se ocupa de devolver la lista de los casos de prueba a partir de la lista `allv` de todos los valores de un tipo de datos. 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.

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, sólo 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 dada un elemento de un tipo cualquiera nos devuelva un entero que representará su tamaño.

Después 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` recibe un valor en su representación no genérica y lo transforma 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`. Finalmente creamos la clase interna `GSized` y definimos la función `gsize`.

Una vez tenemos la interfaz entre las dos clases `Sized` y `GSized` lo siguiente que debemos definir es el constructor sin argumentos, que en nuestro caso devuelve el tamaño 0. A continuacion definimos `size` para un tipo compuesto por otros dos, el tamaño de dicho tipo es la suma de los tamaños de los tipos que los componen. Tras ello 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 del tipo de la derecha y similar si elegimos el constructor de la izquierda. Por último, tenemos la instancia utilizada para trabajar con la metainformación del tipo, que en nuestro caso al no ser necesaria dicha información simplemente llamamos de nuevo a la funcion `gsize` ignorando la metainformación.

```haskell
-- | This is the exported, visible class that inherits from Allv.
class (Allv a) => Sized a where
  -- This function returns the first n elements of size lower or equal m from the "allv" list
  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 :: [a]
  smallest = take uutNumCases (concat $ repeat allv)



  size ::  a -> Int
  default size :: (Generic a, GSized (Rep a)) => a -> Int
  size a = gsize (from a)

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

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

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

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

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

-- | Constants, additional parameters and recursion of kind *
instance Sized a => GSized (K1 i a) where
  gsize (K1 x) = size x
```

Figura 3.1: Clase Sized

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 en orden creciente de tamaos.

Al principio esta clase estaba pensada para ser una única clase que utilizara la librería `Generics` y para contar con un método, `compose` (su funcin se explica ms adelante) 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 a la hora de integrarlo con la clase `Sized` encontramos un problema. La idea que teníamos sobre esta clase 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.

Dadas dos listas la idea es realizar el producto cartesiano de ellas siendo este el resultado de generar todas las parejas con un valor de la primera lista y otro de la segunda. Teniendo en cuenta que ambas pueden ser infinitas, dicho producto deberá ser realizado por diagonales, mostramos la idea en la figura 3.2. La 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 el número de elementos de un tamaño dado siempre es infinito y que siempre hay algún elemento más de tamaño menor o igual a m, aunque est después de muchos elementos intermedios que no lo son. Existe un segundo problema que es el del orden de los constructores, ya que debemos garantizar que en la unin de dos alternativas los casos base se generan antes que los recursivos. Estos problema nos hicieron pensar en utilizar `Template Haskell` en lugar de `Generics`.

En la versión definitiva del programa en el mdulo `TemplateAllv` se encuentra esta funcionalidad de crear una instancia de `Allv` para los tipos de datos definidos por el usuario, utilizando para ello `gen_allv`, con la ayuda de la ya nombrada función `compose` (su cdigo se muestra en la figura 3.3).

La función `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 última y de volver a llamarse a sí misma con los parámetros 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.

Además, dentro de `TemplateAllv` existen tres funciones que se encargan de crear una instancia adecuada de la clase `Allv` para cada uno de los tipos de datos definidos por el usuario.

La primera de ellas, y la más externa en dicho proceso es `gen_allv`, la cual además de llamar a `typeInfo` para extraer la información del tipo y pasarsela a las subfunciones, es también en la que se define, dentro de `gen_body`, como se formará exactamente la nueva función `allv` para la instancia del tipo. Los restantes detalles sobre `gen_allv` pueden verse en el cdigo que se adjunta en el apndice, apartado 8.3.

La siguiente función a tratar, `gen_instance` 8.3 se encarga 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`.

Por último tenemos la función `gen_clause` que es responsable 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 f_1 y 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` sirve para 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 es responsable 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`.

3.4 Instancias predefinidas

En este último apartado mostramos las instancias dentro de las clases `Sized` y `Allv` para los tres tipos básicos (`Int`, `Chary Bool`) y para los tipos que se deducen directamente de ellos, como es el caso de listas de cualquier tipo ya instanciado en dichas clases o las tuplas de hasta longitud 6. (El cdigo correspondiente a dichas instancias se adjunta en el apndice, apartado 8.1)

Como podemos observar en el caso de la instancia en la clase `Sized`, cualquier elemento de uno de los tres tipos tendrá tamaño uno. En el caso de las instancias de los tres tipos en la clase `Allv`, simplemente debemos indicar el conjunto de valores de dicha clase que serán elegibles a la hora de generar casos de prueba, para las cuales utilizamos la función de `compose` explicada con anterioridad.

En el caso de las instancias derivadas dentro de la clase `Sized` éstas se generan mediante `Generics`.

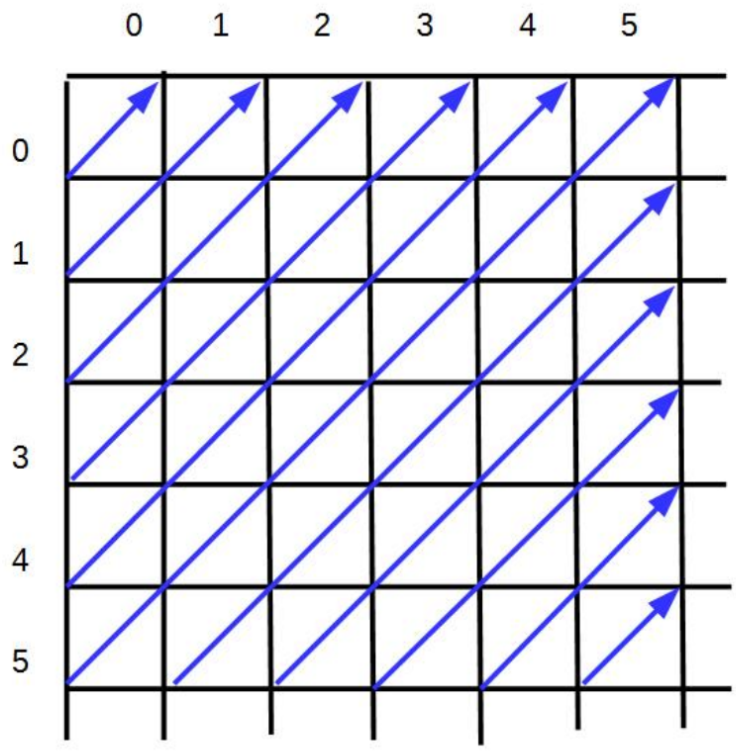

Figura 3.2: Esquema funcionamiento compose

```haskell
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)
```

Figura 3.3: Función compose

Capítulo 4

El generador de casos

4.1 La interfaz con la UUT

En este capítulo vamos a tratar las diferentes fases del proceso de testeo por las que pasa el programa, utilizando para ello un ejemplo de funcionamiento, en este caso una función `insert` en una lista ordenada.

En primer lugar vamos a echar un ojo a la `Unit Under Testing` (a partir de ahora UUT) que se trata de la clase que contiene toda la información sobre la función que vamos a testear en cada momento. Podemos observar la forma que tiene en la figura 4.1.

Este archivo Haskell en el caso de nuestro programa es sintetizado a partir de la función proporcionada por el usuario utilizando la herramienta *IR2Haskell* mencionada en la sección 2.1 y podemos observar que incluye:

- `uutName` indica el nombre de la función a testear para efectos de nombrarla cuando se presentan los resultados al usuario
- Por ultimo las tres funciones `uutPrec`, `uutMethod` y `uutPost` acompañadas de las funciones auxiliares necesarias.

En caso de que las funciones utilizaran algún tipo de datos definido por el usuario su definición se incluiría tambien en el archivo UUT.

4.2 La obtención del tipo de la UUT

El siguiente paso analiza los tipos de los parámetros de entrada de la función que queremos probar, de esa manera podremos generar casos de prueba para dichos tipos de datos. Nos basaremos para ver el proceso en el ejemplo de `insert` comenzado en el apartado anterior.

Este proceso se realiza mediante la función `get_f_inp_types` y sus funciones auxiliares, cuyo código puede encontrarse en la primera parte del apéndice.

`get_f_inp_types` recibe como input una `String` que contiene el nombre de la función a analizar y devolverá al terminar el proceso una lista de `String` que contendrá los tipos de los parámetros de entrada de la función correspondiente al nombre que recibe como entrada. Por ejemplo para la función `insert`.

```
input: "uutPrec"
output: ["Int", "[Int]"]
```

Aquí podemos observar que a su vez la función `get_f_inp_types` se encarga de *monomorfizar* los tipos variables a `Int` lo cual nos permite generar los casos de prueba únicamente para el caso de que la lista sea una lista de enteros y el valor a insertar también sea un entero. Además si obtenemos como resultado de la prueba que la definición de la función es correcta para el caso de los enteros podemos afirmar que lo será para cualquier tipo. El tipo de la función sin monomorfizar sería el siguiente:

```
a -> [a] -> Bool
```

Dentro de `get_f_inp_types` la primera función auxiliar que utilizaremos se trata de `extract_info` la cual recibe como entrada una variable `InfoQ` que es el tipo utilizado en Template Haskell para encapsular toda la información obtenida tras por ejemplo realizar un `reify` como en nuestro caso. `extract_info` analizará la estructura interna de del tipo `InfoQ` y devolverá una terna de tipo `(Name, Name, String)` en la cual el primer elemento se trata del nombre de la función manteniendo el módulo del que procede, el segundo el nombre de la función simplificando el módulo y el tercero una String que contiene el primer análisis sintáctico de los tipos de entrada de la función. Este primer análisis es necesario ya simplificando su resultado conseguiremos la lista de tipos de entrada de la función a analizar y podremos descubrir cuales de ellos no tienen instancias predefinidas en nuestras clases. Este primer análisis tiene la siguiente forma en el caso del `insert` en una lista:

```
-> Int -> [] Int Bool
```

A continuación se llama a la función `simplifyParsing` que recibe como entrada una `String` que se trata del tipo de la función después del primer análisis y devuelve otra `String` con el tipo del análisis simplificado el cual es muy similar a la manera de especificar el tipo de una función en Haskell.

```
Int -> [] _Int -> Bool
```

Finalmente `get_f_inp_types` llama a la función `extract_types` que recibe como entrada una `String` (que será el resultado de la etapa anterior) una lista de `String` (que será la lista en la que vayamos acumulando los tipos encontrados). Esta función devolverá la lista de tipos de la manera que vimos en la salida de `get_f_inp_types`.

4.3 La generación de instancias de Allv y Sized

El tercer paso consiste en generar instancias de las clases `Allv` y `Sized` para todos los tipos de datos que no cuentan con instancias predefinidas, en nuestro caso solo hay instancias para `Int`, `Char` y `Bool` junto con las listas y las tuplas de hasta longitud 6.

Ejemplificaremos dicho proceso para el caso de `insert` y su instancia en la clase `Allv` ya que la creación de la instancia de `Sized` sigue un proceso similar.

La función más externa en este proceso es `gen_allv_str_listQ` recibe como entrada una lista de `String` que contiene los tipos necesarios para la prueba que no cuentan con instancias predefinidas de `Allv` y devuelve como resultado una lista de `Dec` que es el tipo utilizado en Template Haskell para representar declaraciones. La llamada a `gen_allv_str_listQ` se realiza desde el módulo `UUTReader` dentro de un *splice* que será el encargado de traducir la lista de `DecQ` en instancias de Haskell y presenta el siguiente aspecto:

```
$(gen_allv_str_listQ (notDefTypesQMonad (get_f_inp_types (head uutMethods))))
```

Tras extraer la lista de `String` de la mónada `Q` nos encontramos con la función `gen_allv_str` la cual recibe como entrada el nombre de un único tipo de datos y devuelve la declaración generada para él. Para llegar a dicha declaración `gen_allv_str` realiza un analisis en profundidad sobre el tipo de datos y sus constructoras:

- En primer lugar reordena las constructoras de datos, de manera que siempre sean ejecutadas en primer lugar las no recursivas, ya que las si recursivas dependen de las primeras.
- Analiza para cada una de las constructoras de tipos cuales son los tipos de sus parámetros.
- Tiene en cuenta el número de parámetros de cada una de las constructoras. Aquellas que tengan dos o más parámetros tendrán que ser llamadas mediante una función auxiliar que será aplicada a la lista de tuplas devuelta por la función `compose`.

Un ejemplo de una instancia generada mediante `gen_allv_str_listQ` sería el siguiente, equivalente al tipo de datos `Tree` utilizado en los experimentos 3 y 4.

```
 instance Allv x1 => Allv (Tree x1)
where allv = (GHC.Base.++) [UUT.Empty]
             (GHC.Base.map f2 (Sized.compose Sized.allv (Sized.compose Sized.allv Sized.allv)))
  where f2 (x3, (x2, x1)) = UUT.Node x3 x2 x1
```

4.4 La generación y ejecución de casos

Por último vamos a ver como se generan los casos de prueba y se ejecutan para comprobar si la definición de la función bajo prueba es correcta o no. Esta parte del proceso está dirigida por la función `test`, que no recibe ningún parámetro pues recoge la información necesaria para su llamada directamente de la UUT y devuelve como resultado una terna del tipo `([Bool],[a],[a])` donde a es el tipo de los datos de entrada de la función a probar.

- La primera lista de la terna contiene una lista que indica para cada uno de los casos generados que cumplen la precondición si tras ejecutarlo también cumple la postcondición.
- La segunda es la lista de todos los casos generados, antes de comprobar cuales de ellos cumplen la precondición.
- La última lista son todos los casos de prueba de los generados que cumplen la precondición.

Dentro de la función `test` nos encontramos la llamada a `smallest` la cual ademas se realiza especificando de que tipo de datos queremos que sea dicha lista generada por `smallest`. Esta llamada nos devolverá la lista de los 1000 menores casos de prueba generados para el tipo del input de la función. El número de casos de prueba es una constante que puede ser modificada por el usuario. Dicha llamada es la siguiente:

```
smallest :: $(inputT (head uutMethods))
```

Una vez generados estos casos de prueba los usaremos en la llamada a `prueba`, la cual recibe como entrada la lista de casos de prueba y aplica sobre ellos la precondición, la función y la postcondición y devuelve como resultado la misma terna de listas devuelta por `test` y comentada con anterioridad. Dicha ejecución de los casos de prueba se realiza en las tres fases lógicas:

- En primer lugar se filtra la lista de todos los casos de prueba mediante la función de la precondición para quedarnos solo con aquellos que la cumplen.
- En segundo lugar sobre la lista de los casos que cumplen la precondición aplicamos la función que queremos probar, para de esta manera conseguir para cada caso su output correspondiente.
- Finalmente llamamos a la postcondición utilizando para ello las dos listas generadas anteriormente: la de los casos de prueba que cumplen la precondición y la de los outputs de los casos de prueba. De esta manera podremos saber si algun caso falla y notificarlo al usuario.

Los resultados de la prueba serán presentados al usuario mediante la función `printInfoTuple` que recibe como entrada la terna de tipo `([Bool],[a],[a])` e imprime en pantalla:

- Número de casos generados y cuantos de ellos cumplen la precondición.
- Todos los casos de prueba generados.
- Todos los casos generados que cumplen la precondición.
- Los casos que no cumplen la postcondición en caso de que exista alguno.

```haskell
module UUT where

import qualified Arrays as A
import qualified Bags as B
import qualified Sets as S
import qualified Sequences as Q
import Assertion
import Data.List

uutNargs :: Int
uutNargs = 2

uutMethods :: [String]
uutMethods = ["uutPrec", "uutMethod", "uutPost"]

uutName :: String
uutName = "insert"

uutPrec :: Int -> [Int] -> Bool
uutPrec x xs = sorted xs

sorted []       = True
sorted [x]      = True
sorted (x:y:xs) = x <= y && sorted (y:xs)


uutMethod :: Int -> [Int] -> [Int]
uutMethod x [] = [x]
uutMethod x (y:ys) | x <= y    = x:y:ys
                   | otherwise = y : uutMethod x ys

uutPost :: Int -> [Int] -> [Int] -> Bool
uutPost x xs ys = ys == sort (x:xs)
```

Figura 4.1: UUT para insert en una lista ordenada

Capítulo 5

Experimentos

El código UUTs utilizadas en estos experimentos pueden ser encontradas en la sección 8.5 dentro del apéndice.

5.1 Insertar un elemento en una lista

Función: insert1 x xs
Precondición: $Prec(x, xs) = sorted(xs)$ que puede expresarse como $\forall i, j.0 \leq i \leq j < length(xs) \rightarrow xs!i \leq xs!j$
Postcondición: $Post(x, xs, res) = sorted(res) permut(x : xs, res)$ donde permut(xs,ys) expresa que una lista es una permutacion de la otra

Se generaron para probar dicha función un total de 1000 casos de prueba, de los cuales pasaron la precondición un total de 518 casos de prueba. De todos esos casos de prueba que pasaron la precondición todos ellos pasaron la postcondición, no encontramos casos que la contradijeran.

En la figura 5.1 podemos ver la forma que tienen los primeros casos de prueba de aquellos que pasaron la precondición.

5.2 Insertar un elemento en un Array

Función: insert2 x m a
Precondición: $Prec(x, m, a) = 0 \leq m < length(a) sorted(a, 0, m - 1)$ donde sorted (a,i,j) expresa que el array está ordenado entre i y j
Postcondición: $Post(x, m, a, res) = sorted(a, 0, m)$

Se generaron para probar dicha función un total de 1000 casos de prueba, de los cuales pasaron la precondición un total de 157 casos de prueba. De todos esos casos de prueba que pasaron la precondición todos ellos pasaron la postcondición, no encontramos casos que la contradijeran.

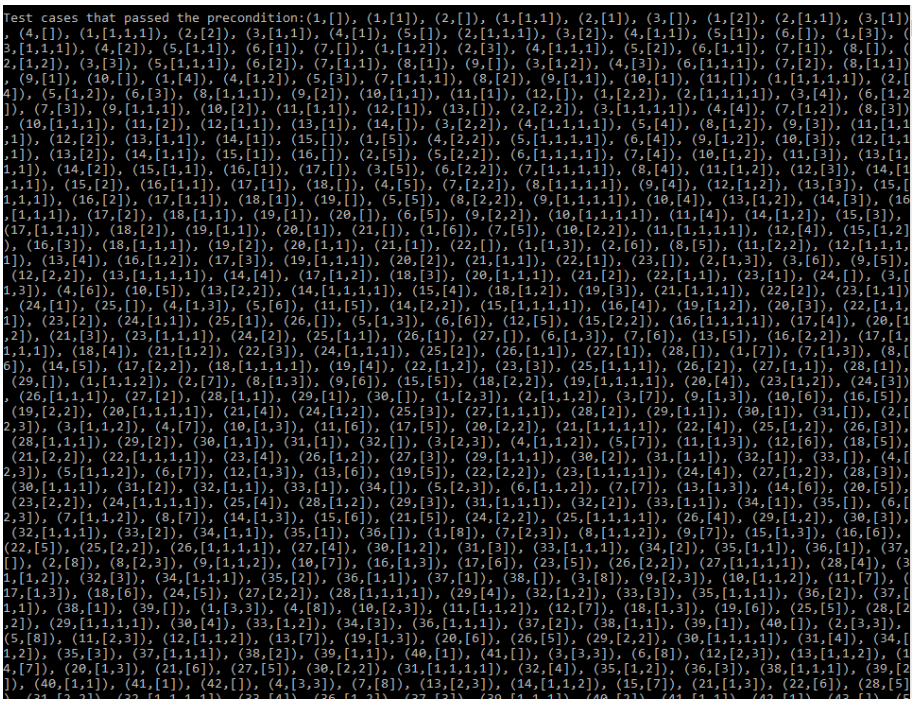

Figura 5.1: Casos de la primera prueba que pasaron la precondición

5.3 Insertar un elemento en un árbol

Función: insertBST x
Precondición: $Prec(x,t) = sorted(inorder(t))$ es decir, la propiedad de ser un árbol de búsqueda
Postcondición: $Post(x,t,res) = sorted(inorder(res))permut(x : inorder(t), inorder(res))$, es decir ambas listas tienen los mismos elementos

Se generaron para probar dicha función un total de 1000 casos de prueba, de los cuales pasaron la precondición un total de 518 casos de prueba. De todos esos casos de prueba que pasaron la precondición todos ellos pasaron la postcondición, no encontramos casos que la contradijeran.

5.4 Búsqueda en un árbol

Función: search x t
Precondición: $Prec(x,t) = sorted(inorder(t))$ es decir, la propiedad de ser un árbol de búsqueda
Postcondición: $Post(x,t,res) = res \leftrightarrow x \in inorder(t)$ es decir, el resultado es cierto

si, y solo si, x pertenece al árbol t

Se generaron para probar dicha función un total de 1000 casos de prueba, de los cuales pasaron la precondición un total de 518 casos de prueba. De todos esos casos de prueba que pasaron la precondición 45 de ellos no pasaron la postcondición.

5.5 Conclusiones de los experimentos

En las cuatro pruebas podemos observar que de los 1000 ejemplos generados, en todos ellos un porcentaje razonable pasa la precondición, incluso en el segundo caso que es el que cuenta con una precondición más fuerte.

En los tres casos en los cuales la definición de la función, su precondición y postcondición son correctas nuestro programa no detecta ningun error, todos lo casos de prueba que cumplen la precondición son aceptados como correctos, en cambio en el último de los casos, el cual fue definido incorrectamente a propósito el programa detecta que está definido incorrectamente con una buena cantidad de contraejemplos, cerca de un 10% de los casos que pasaron la precondición.

Capítulo 6

Trabajo relacionado y conclusiones

6.1 Korat

La primera de las herramientas que vamos a tratar en este apartado es **Korat** [1], una herramienta de Java que sirve para la generación de casos complejos de prueba a partir de unas restricciones dadas.

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

Lo primero que hace **Korat** es reservar el espacio necesario para los objetos especificados, en el caso de un `BinTree` reservaria espacio para él y para el número de Nodos que queramos. Por ejemplo, si queremos un árbol 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`.

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 dentro de cada uno de los dominios de los campos será consistente con el 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.

Tras esto, cada candidato de la entrada se respresenta como un vector de índices de sus correspondientes dominios de campos.

Tras definir los dominios de cada uno de los campos del vector comienza la busqueda con la inicialización a 0 de todos los indices del vector. A continuación fijamos los valores de los campos para cada posible candidato 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. Durante dicha ejecución **Korat** monitoriza el orden en que 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 índice del dominio de campo para el campo que se encuentra último en la lista ordenada construida previamente. Si dicho índice es mayor que el tamaño del dominio de su campo, este se pone a cero y se incrementa el índice de la posición anterior y así sucesivamente. Al seguir este método para generar el siguiente candidato conseguiremos podar un gran número de ellos que tienen la misma evaluación parcial sin dejar fuera ninguno válido.

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 son generados todos los candidatos para los que `repOk` devuelve True. Los casos para los que siempre devuelve False nunca son generados y los casos para los que alguna vez se devuelve True y otras veces False pueden ser generados o no.

Dos candidatos serán definidos como isomorfos si las partes de sus grafos alcanzables desde la raíz son isomorfas. En el caso de `repOk` el objeto raíz es aquel 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 generar 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` [7] 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.

La principal diferencia de `Smallcheck` respecto a `Quickcheck` es la forma en que genera 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`, que 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 continuación presentaremos 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 de todos sus argumentos.
- En el caso de las tuplas, dicha profundidad se define de manera un poco diferente. La profundidad de una tupla de aridad cero es cero pero la de una tupla de aridad positiva es la mayor profundidad de entre todas las de sus componentes.
- En el caso de los tipos numéricos, la definición de la profundidad se realiza 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 $Succ^i Zero$. A su vez, la profundidad de un numero decimal $sx2^e$ es la de la tupla de enteros (s,e).

`Smallcheck` define una clase `Serial` de tipos que pueden ser enumerados hasta una determinada profundidad. Existen instancias predefinidas de la clase `Serial` para todos los tipos de datos del preludio . Sin embargo, es muy fácil definir una nueva instancia de dicha clase para un tipo de datos algebraico, ésta es de un conjunto de combinadores `cons<N>`, genéricos para cualquier combinación de tipos Serial, donde **N** es la aridad del constructor.

Supongamos un tipo de datos en Haskell `Prop` en el que tenemos una variable, la negación de una variable y el `Or` de dos variables.

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

Para dicho tipo de datos definir una instancia de la clase `Serial`, asumiendo una definición similar para el tipo `Name`, sería.

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

Una serie es simplemente 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<N>` están definidos usando `><` decrementando y comprobando la profundidad correctamente.

```
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)]
```

Cuando se usa muchas veces el esquema general para definir valores de prueba se produce que para alguna profundidad pequeña `d` los 10.000-100.000 casos de prueba son comprobados rápidamente, 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 búsqueda 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 base para las pruebas, debemos considerar que los espacios de busqueda para los tipos compuestos (especialmente funcionales) al usar bases numéricas, crecen de manera muy rápida. En muchos casos el tipo booleano puede ser una elección perfectamente válida para los valores base, y con ello se conseguiróa reducir en gran medida el espacio de busqueda respecto a la utilización de enteros.

Existe otra versión de `Smallcheck` llamada `Lazy Smallcheck`, 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 obtener 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 la misma. 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.

Capítulo 7

Conclusiones del proyecto

7.1 Conclusiones

La idea de generar valores para tipos de datos predefinidos y definidos por el usuario mediante el uso de las clases de Haskell ya está presente tanto de `Quickcheck` como en `Smallcheck`. La idea de generar casos de prueba exhaustivos hasta un cierto tamaño, también está presente tanto en `Smallcheck` como en **Korat**. La diferencia principal de nuestro trabajo con estos es que los tres requieren que el usuario escriba código adicional para los tipos del usuario que son desconocidos para el sistema. En el caso de `Quickcheck`, hay que generar manualmente la instancia de la clase Arbitrary, si bien el sistema ofrece una serie de combinadores que facilitan la tarea. En el caso de `Smallcheck`, hay que escribir manualmente una instancia de la clase `Serial`, y en el caso de **Korat** hay que editar una plantilla para definir una noción de tamaño y para evitar generar valores duplicados.

En nuestro trabajo, tanto la noción de tamaño, como las instancias de la clases Allv y Sized, se generan automáticamente para los tipos desconocidos, gracias al uso de respectivamente Template Haskell y `Generics`. Ello unido a que el código de la precondicion y la postcondición son generados automáticamente por la herramienta previa IR2Haskell, hace que toda el proceso de prueba desde que el usuario escribe su código y asertos originales, hasta que se ejecutan las pruebas y se detectan los posibles errores, se haga sin ninguna intervención manual.

Durante la creación de los experimentos reportados en este trabajo, la herramienta fue capaz de detectar errores no intencionados, tanto en las postcondiciones inicialmente escritas, como en el código bajo prueba, lo cual a la vez sirvió para asegurarnos de que la herramienta detecta de manera correcta errores en la definición de la función.

7.2 Conclusions

The idea of generating values both for the predefined datatypes and the types defined by the user using Haskell classes is already present both in `Quickcheck` and

`Smallcheck`. The idea about genrating exhaustive test cases to a certain size is also present both in `Smallcheck` and **Korat**. The main difference between our project and all those projects are that those three need the user to write additional code for the user-defined datatypes that are not know by the system. In `Quickcheck` is necessary to manually generate instance for the Arbitrary class using a set of combiners given to do so. In the case of `Smallcheck` it's necessary to manually create an instance for the `Serial` class and in **Korat** you have to edit a template to define the concept of size and being able to avoid duplicated values.

In our project, both the concept of size and the instances of Allv and Sized classes are automatically generated for all the unknown data types thanks to Template Haskell and `Generics`. This together with the fact that the code for the precondition and postcondition are automatically generated by the tool IR2Haskell makes all the testing process, from the moment in which the user writes its code and asserts until the moment in which tests are executed and the possible errors are detected flow without any intervention from the user.

During the creation of the experiments shown in this project, the tool was able to find some non intended errors both in some firstly writen postconditions and code under test, which served us to be completely sure about the correct functioning of the tool as it was able to detect incorrectness in the function definition

Capítulo 8

Apéndice

8.1 Instancias predefinidas

```
-- | For basic types we must give the instances
instance Sized Int where
size x = 1

instance Allv Int where
allv = [1..5]

instance Sized Char where
size x = 1

instance Allv Char where
allv = ['a'..'z']

instance Sized Bool where
size x = 1

instance Allv Bool where
allv = [True, False]

instance Sized a => Sized [a]

instance (Sized a, Sized b) => Sized (a,b)

instance (Sized a, Sized b, Sized c) => Sized (a,b,c)

instance (Sized a, Sized b, Sized c, Sized d) => Sized (a,b,c,d)

instance (Sized a, Sized b, Sized c, Sized d, Sized e) => Sized (a,b,c,d,e)
```

```
instance (Sized a, Sized b, Sized c, Sized d, Sized e, Sized f) => Sized (a,b,c,d,e,f)

instance Allv a => Allv [a] where
allv = [] : (map (\(x,xs) -> x:xs) $ compose allv allv)

instance (Allv a, Allv b) => Allv (a,b) where
allv = compose allv allv

instance (Allv a, Allv b, Allv c) => Allv (a,b,c) where
allv = map (\(x,(y,z)) -> (x,y,z)) (compose allv (compose allv allv))

instance (Allv a, Allv b, Allv c, Allv d) => Allv (a,b,c,d) where
allv = map (\(x,(y,(z,t))) -> (x,y,z,t)) (compose allv (compose allv (compose allv allv)))

instance (Allv a, Allv b, Allv c, Allv d, Allv e) => Allv (a,b,c,d,e) where
allv = map (\(x,(y,(z,(t,u)))) -> (x,y,z,t,u)) (compose allv (compose allv
             (compose allv (compose allv allv))))

instance (Allv a, Allv b, Allv c, Allv d, Allv e, Allv f) => Allv (a,b,c,d,e,f) where
allv = map (\(x,(y,(z,(t,(u,v))))) -> (x,y,z,t,u,v)) (compose allv (compose allv
             (compose allv (compose allv (compose allv allv)))))
```

8.2 Obtención del tipo de la UUT

```
----------------Get types for the input params---------------------
get_f_inp_types :: String -> Q [String]
get_f_inp_types str = do (Just name) <- lookupValueName str
(_,_,text) <- extract_info (reify name)
t <- return(simplifyParsing text)
return (extract_types t [] "")


--------Auxiliar functions for getFInpTypes ------------------------

extract_info :: InfoQ -> Q(Name, Name, String)
extract_info m =
do d <- m
case d of
d@(VarI _ _ _ _) ->
return $ (funcName d, simpleName $ funcName d, parseDataTypes d)
d@(ClassOpI _ _ _ _) ->
return $ (funcName d, simpleName $ funcName d, parseDataTypes d)
_ -> error ("Error in extract_info" ++ show d)
where
funcName (VarI n _ _ _)  = n
funcName (ClassOpI n _ _ _) = n
parseDataTypes (VarI _ x _ _) = first_parse x
```

```
parseDataTypes (ClassOpI _ x _ _) = first_parse x
first_parse ((ForallT _ _ x)) = parsing x
first_parse x = parsing x
parsing ((AppT (ConT x) (VarT y))) = "{" ++ (parsing (ConT x)) ++
   "_" ++ (parsing (VarT y)) ++ "}"
parsing ((AppT (ConT x) (ConT y))) = "{" ++ (parsing (ConT x)) ++
   "_" ++ (parsing (ConT y)) ++ "}"
parsing ((AppT x y)) = (parsing x) ++ (parsing y)
parsing (ArrowT) = "-> "
parsing (ListT) = "[] "
parsing ((TupleT 2)) = "(,) "
parsing ((TupleT 3)) = "(,,) "
parsing ((VarT _)) = "Int "
parsing ((ConT x)) = (nameBase x) ++ " "
parsing _ = "UND "


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

eliminateMaybe :: Maybe a -> a
eliminateMaybe (Just a) = a

simplifyParsing :: String -> String
simplifyParsing string = fst (auxiliarParse string)

auxiliarParse s
| startswith "->" (lstrip s) = ( (fst call2) ++ " -> " ++
   (fst (call 0 (snd call2))) , snd (call 0 (snd call2)))
| startswith "[]" (lstrip s) = ("[]_" ++ (fst call2) , snd call2)
| startswith "(,)" (lstrip s) = ("(,)_" ++ (fst call3) ++
   (fst (call 0 (snd call3))) , snd (call 0 (snd call3)))
| startswith "(,,)" (lstrip s) = ("(,,)_" ++ (fst call4) ++
   (fst (call 0 (snd call4))) ++(fst (call 0 (snd (call 0 (snd call4)))))
     , snd (call 0 (snd (call 0 (snd call4)))))
| startswith "(,,,)" (lstrip s) = ("(,,,)_" ++ (fst call5) ++
   (fst (call 0 (snd call5))) ++ (fst (call 0 (snd (call 0 (snd call5))))) ++
     (fst (call 0 (snd (call 0 (snd (call 0 (snd call5)))))))) ,
       snd (call 0 (snd (call 0 (snd (call 0 (snd call5)))))))
| startswith "{" (lstrip s) = compoundVar (lstrip s) ""
| otherwise = baseVar (lstrip s) ""

where call2 = call 2 s
call3 = call 3 s
```

```
call4 = call 4 s
call5 = call 5 s
call n string = auxiliarParse (stringSkip n (lstrip string))
stringSkip n (x:xs)
| n == 0 = (x:xs)
| otherwise = stringSkip (n-1) xs
baseVar (x:xs) formedS
| x /= ' '  = baseVar xs (formedS ++ [x])
| otherwise = (formedS , (lstrip xs))
compoundVar (x:xs) formedS
| x == '}' = (formedS , (lstrip xs))
| x == '{' = compoundVar xs formedS
| otherwise = compoundVar xs (formedS ++ [x])

-------------Extracts the input types from the simplify parsing-----

extract_types :: String -> [String] -> String -> [String]
extract_types [] list _ = list
extract_types (x1:xs) list building_t
| (x1 == ' ') || (x1 == '>') = extract_types xs list building_t
| (x1 == '-') = extract_types xs (list++[building_t]) ""
| (x1 == '_') = extract_types xs list (building_t ++[' '])
| otherwise = extract_types xs list (building_t++[x1])
```

8.3 Generación de instancias de Allv

```
gen_allv_str :: String -> Q Dec
gen_allv_str str = do
(typeStr, numArg) <- return (headArgs(split_str str ""))
(Just typeName) <- lookupTypeName typeStr
gen_allv typeName numArg
where
headArgs :: [String] -> (String,Int)
headArgs (x:xs) = (x, (length xs))
split_str :: String -> String -> [String]
split_str [] saved = [saved]
split_str (x:xs) saved
| x == ' ' = saved:(split_str xs "")
|otherwise = split_str xs (saved++[x])



gen_allv_list :: Name -> Int -> Q [Dec]
gen_allv_list name n = do
dec <- gen_allv name n
return [dec]
```

```
-- Generate an intance of the class Allv for the type typName
gen_allv :: Name -> Int -> Q Dec
gen_allv t n =
do (TyConI d) <- reify t
--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") t_name cInfo consts
typesCons noSimplifiedName (mkName "allv", gen_body) n
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 = [listExps] ++ (gen_body is cs fs)
| otherwise = [appsE (varE '(++):[listExps] ++ gen_body is cs fs)]
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.
listExps = if(i > 0) then appsE (mapE:constructorF:(allvFunc i))
else listE [appsE (constructorF:[])]
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 = appsE [varE 'allv]
moveHead (x1:x2:xs) = x2:x1:xs
allvFunc 0 = []
allvFunc 1 = [allvE]
allvFunc n = [appsE (composeE:[allvE] ++ allvFunc (n-1))]



--Construct an instance of class class_name for type for_type
--with a corresponding function  to build the method body
gen_instance :: Name -> Name -> [Int] -> [Name]
```

```
-> [[Type]] -> Name -> Func -> Int -> DecQ
gen_instance class_name for_name cInfo consts typesCons typeName_nosimp func n =
instanceD (cxt (map applyConst ctxTypes))
(appT (conT class_name) (foldl appT (conT for_name) (map varT ctxTypes)))
[(func_def func)]
where func_def (func_name, gen_func) = funD func_name
[gen_clause gen_func cInfo consts typesCons typeName_nosimp]
applyConst var_name = appT (conT class_name) (varT var_name)
ctxTypes :: [Name]
ctxTypes = map (\ x -> mkName ("x"++ (show x))) (take n [1..])


-- 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)]
```

8.4 Generación y ejecución de casos

```
test = prueba listArgs
where listArgs = (smallest :: $(inputT (head uutMethods)))

--------------------Prueba function-------------------------------------
prueba listArgs = ((pos_f filtered_pre output), (listArgs), (filtered_pre))
where
filtered_pre = pre_f listArgs
output = fun_f filtered_pre

pre_f listArgs = filter (prec_f_aux) listArgs

fun_f filtered_list = map fun_f_aux filtered_list

pos_f inputs outputs = zipWith pos_f_aux inputs outputs

--------------generators for auxiliar prueba functions-------------------------
prec_f_aux $(tupleP uutNargs) = $(appsE ((varE 'uutPrec):(map varE (listVar uutNargs))))

fun_f_aux $(tupleP uutNargs) = $(appsE ((varE 'uutMethod):(map varE (listVar uutNargs))))

pos_f_aux $(tupleP uutNargs) $(varP $ mkName "o") = $(appsE ((varE 'uutPost):
     ((map varE (listVar uutNargs))++[varE $ mkName "o"])))
```

8.5 UUTs de los diferentes casos de prueba

8.5.1 Insertar en una lista ordenada

```
module UUT where

import qualified Arrays as A
import qualified Bags as B
import qualified Sets as S
import qualified Sequences as Q
import Assertion
import Data.List

uutNargs :: Int
uutNargs = 2

uutMethods :: [String]
uutMethods = ["uutPrec", "uutMethod", "uutPost"]

uutName :: String
uutName = "insert"

uutPrec :: Int -> [Int] -> Bool
uutPrec x xs = sorted xs

sorted []       = True
sorted [x]      = True
sorted (x:y:xs) = x <= y && sorted (y:xs)


uutMethod :: Int -> [Int] -> [Int]
uutMethod x [] = [x]
uutMethod x (y:ys) | x <= y    = x:y:ys
   | otherwise = y : uutMethod x ys

uutPost :: Int -> [Int] -> [Int] -> Bool
uutPost x xs ys = ys == sort (x:xs)
```

8.5.2 Insertar en un Array

```
module UUT where

import qualified Arrays as A
import qualified Bags as B
import qualified Sets as S
import qualified Sequences as Q
import Assertion
```

```
import Data.List

uutNargs :: Int
uutNargs = 3

uutMethods :: [String]
uutMethods = ["uutPrec", "uutMethod", "uutPost"]

uutName :: String
uutName = "insert" --TODO usar al principio del output

uutPrec x m a = evalA $
And
(FTerm (Aplic
(Aplic
(TVar (<=))
((TConst 0)))
((TVar m))))
(And
(FTerm (Aplic
(Aplic
(TVar (<))
((TVar m)))
(Aplic
((TVar A.len)) ((TVar a)))))
(Forall
(GuardIntTuple
(Tuple2 ((TConst 0)) ((TConst 0)))
(Tuple2 (Aplic (Aplic (TVar (-)) (TVar m))
(TConst 1)) (Aplic (Aplic (TVar (-)) (TVar m)) (TConst 1))))
(\(i, j) -> (Imp
(FTerm (Aplic
(Aplic
(TVar (<=))
((TConst 0)))
((TVar i))))
(Imp
(FTerm (Aplic
(Aplic
(TVar (<=))
((TVar i)))
((TVar j))))
(Imp
(FTerm (Aplic
(Aplic
(TVar (<))
((TVar j)))
((TVar m))))
(FTerm (Aplic
```

```
(Aplic
(TVar (<=))
(Aplic
(Aplic
((TVar A.get))
((TVar a))) ((TVar i))))
(Aplic
(Aplic
((TVar A.get))
((TVar a))) ((TVar j)))))))))))

uutMethod x m a =
let i = (-) m 1 in
f2 x m i a
where
f2 x m i a =
let b1 = (>=) i 0 in
case b1 of
False -> f4 x m i a
True -> let e = A.get a i in
let b2 = (<) x e in
case b2 of
True -> let e = A.get a i in
let i2 = (+) i 1 in
let ap = A.set a i2 e in
let i3 = (-) i 1 in
f2 x m i3 ap
False -> f4 x m i a
f4 x m i a =
let i2 = (+) i 1 in
let ap = A.set a i2 x in
ap

uutPost x m a res = evalA $
Forall
(GuardIntTuple
(Tuple2 ((TConst 0)) ((TConst 0)))
(Tuple2 ((TVar m)) ((TVar m))))
(\(i, j) -> (Imp
(FTerm (Aplic
(Aplic
(TVar (<=))
((TConst 0)))
((TVar i))))
(Imp
(FTerm (Aplic
(Aplic
(TVar (<=))
((TVar i)))
```

```
((TVar j))))
(Imp
(FTerm (Aplic
(Aplic
(TVar (<=))
((TVar j)))
((TVar m))))
(FTerm (Aplic
(Aplic
(TVar (<=))
(Aplic
(Aplic
((TVar A.get)) ((TVar res))) ((TVar i))))
(Aplic
(Aplic
((TVar A.get)) ((TVar res))) ((TVar j)))))))))
```

8.5.3 Insertar en un árbol

```
-- This file has been generated by the CAVI-ART CLIR-to-Haskell transformer tool

-# LANGUAGE DeriveGeneric #-
module UUT where


import qualified Arrays as A
import qualified Bags as B
import qualified Sets as S
import qualified Sequences as Q
import Assertion
import Data.List
import GHC.Generics

-- Inserting in a Binary Search tree
-- This is an example where the user defines a new type

data Tree a = Empty
| Node (Tree a) a (Tree a)
deriving (Generic,Show,Eq)


uutNargs :: Int
uutNargs = 2

uutMethods :: [String]
uutMethods = ["uutPrec", "uutMethod", "uutPost"]

uutName :: String
```

```
uutName = "insertBST"

uutPrec :: Int -> Tree Int -> Bool
uutPrec x t = sorted $ inorder t

inorder Empty        = []
inorder (Node l x r) = inorder l ++ (x : inorder r)

sorted []       = True
sorted [x]      = True
sorted (x:y:xs) = x <= y && sorted (y:xs)


uutMethod :: Ord a => a -> Tree a -> Tree a
uutMethod x Empty = Node Empty x Empty
uutMethod x t@(Node l y r)
| x < y  = Node (uutMethod x l) y r
| x == y = t
| x > y  = Node l x (uutMethod x r)

uutPost x t o = if x `elem` inorder t then t == o else inorder o == sort (x : inorder t)
```

8.5.4 Búsqueda en un árbol

```
-# LANGUAGE DeriveGeneric #-
module UUT where


import qualified Arrays as A
import qualified Bags as B
import qualified Sets as S
import qualified Sequences as Q
import Assertion

import GHC.Generics

-- Searching in a Binary Search tree
-- This is an example where the user defines a new type

data Tree a = Node (Tree a) a (Tree a)
| Empty
deriving (Generic,Show)


uutNargs :: Int
```

```
uutNargs = 2

uutMethods :: [String]
uutMethods = ["uutPrec", "uutMethod", "uutPost"]

uutName :: String
uutName = "searchBST"


uutPrec :: Int -> Tree Int -> Bool
uutPrec x t = sorted $ inorder t



inorder Empty        = []
inorder (Node l x r) = inorder l ++ (x : inorder r)

sorted []       = True
sorted [x]      = True
sorted (x:y:xs) = x <= y && sorted (y:xs)


uutMethod :: Ord a => a -> Tree a -> Bool
uutMethod x Empty = False
uutMethod x t@(Node l y r)
| x < y  = uutMethod x r -- error, debería ser l
| x == y = True
| x > y  = uutMethod x r

uutPost x t o = o == (x `elem` inorder t)
```

Bibliografía

[1] Chandrasekhar Boyapati, Sarfraz Khurshid & Darko Marinov (2002): *Korat: Automated Testing Based on Java Predicates*. Available at `http://web.eecs.umich.edu/ bchandra/publications/issta02.pdf`.

[2] Koen Claessen & AJohn Hughes (2000): *QuickCheck:A Lightweight Tool for Random Testing of Haskell Programs*. Available at `https://www.eecs.northwestern.edu/ robby/courses/395-495-2009-fall/quick.pdf`.

[3] Moreno Falaschi, editor (2015): *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*. *Lecture Notes in Computer Science* 9527, Springer, doi:10.1007/978-3-319-27436-2. Available at `http://dx.doi.org/10.1007/978-3-319-27436-2`.

[4] Magalhaes, Atze Dijkstra, Johan Jeuring & Andres Lh (2010): *A Generic Deriving Mechanism for Haskell*. Available at `http://www.dreixel.net/research/pdf/gdmh_nocolor.pdf`.

[5] Manuel Montenegro, Susana Nieva, Ricardo Peña & Clara Segura (2016): *Extending Liquid Types to Arrays*. In: *PROLE 2016, Salamanca, Spain*, pp. 1–15.

[6] Manuel Montenegro, Ricardo Peña & Jaime Sánchez-Hernández (2015): *A Generic Intermediate Representation for Verification Condition Generation*. In: *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, pp. 227–243.

[7] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *SmallCheck and Lazy SmallCheck automatic exhaustive testing for small values*. Available at `https://pdfs.semanticscholar.org/2460/c9b40ea3c4bbaef53c5f4ad2717154cf15b5.pdf`.

[8] Tim Sheard & Simon Peyton Jones (2002): *Template Meta-programming for Haskell*. Available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meta-haskell.pdf`