Modelando Generadores de Datos Aleatorios Mediante Procesos Estocásticos

Tesina de grado presentada por
Claudio Agustín Mista
al
Departamento de Ciencias de La Computación
de la Universidad Nacional de Rosario

en cumplimiento parcial de los requerimientos
para la obtención del grado de
Licenciado en Ciencias de la Computación

bajo la dirección de Alejandro Russo
y la co-dirección de Martín Ceresa

Abril de 2018
Rosario, Argentina

Modelando Generadores de Datos Aleatorios Mediante Procesos Estocásticos

Resumen

El testing aleatorio de software es una de las técnicas de verificación más prolíferas de los últimos años. En la comunidad de Haskell, *QuickCheck* es sin lugar a dudas la herramienta de testing aleatorio más influyente. El desarrollador describe las propiedades que su sistema debe verificar, y *QuickCheck* luego se encarga de generar un gran número de datos aleatorios buscando vulnerar alguna de ellas. Este proceso usualmente requiere que el desarrollador escriba manualmente generadores de datos aleatorios para los tipos de datos involucrados en las propiedades del sistema. Si bien este proceso puede ser automatizado utilizando herramientas de meta-programación, los resultados obtenidos carecen de la robustez requerida muchas veces en la práctica.

En este trabajo se provee un formalismo matemático basado en procesos estocásticos capaz de predecir la distribución de datos generados aleatoriamente mediante generadores escritos usando *QuickCheck*. A partir del mismo, se presenta un mecanismo de obtención automática de generadores aleatorios, junto a una heurística capaz de optimizar los parámetros de generación, con el fin de ajustar la distribución de los datos generados a las demandas del desarrollador. Junto a esto, se provee una implementación en Haskell de nuestras ideas, con la cual realizamos experimentos sobre sistemas de software existentes de diversa complejidad. Nuestros resultados indican mejoras importantes en el cubrimiento de código obtenido respecto de las herramientas presentes en el estado del arte en materia de testing aleatorio automático en Haskell.

Este trabajo se encuentra basado en el artículo "Branching Processes for QuickCheck Generators" por Agustín Mista, Alejandro Russo y John Hughes; el cual fue enviado al ACM SIGPLAN Haskell Symposium 2018, y que se encuentra actualmente bajo revisión.

Índice general

1. Introducción 3

2. Preliminares 7
- 2.1. Haskell . . . 7
- 2.2. Template Haskell . . . 12
- 2.3. QuickCheck . . . 15

3. Random Testing Automático en Haskell 21
- 3.1. *derive* . . . 21
- 3.2. *MegaDeTH* . . . 25
- 3.3. Un enfoque distinto: *Feat* . . . 28

4. Marco Teórico 31
- 4.1. Procesos de Ramificación de Tipo Simple . . . 32
- 4.2. Procesos de Ramificación Multi-Tipo . . . 37
- 4.3. Predicción de Constructores Terminales . . . 43
- 4.4. Predicción de Tipos de Datos Mutuamente Recursivos . . . 47
- 4.5. Matriz de Reproducción de Tipos . . . 49
- 4.6. Predicción de Tipos de Datos Compuestos . . . 52

5. Evaluación 55
- 5.1. Implementación . . . 56
  - 5.1.1. Proceso de derivación . . . 57
  - 5.1.2. Funciones de costo . . . 59
- 5.2. Casos de Prueba . . . 63

6. Conclusiones 69
- 6.1. Trabajo Futuro . . . 71

Apéndice A. Demostraciones 73

Referencias 84

1
Introducción

El testing de software es un proceso que busca proveer información relevante acerca de la *calidad* de una componente o sistema de software. Este proceso esencialmente involucra la ejecución del mismo ante un entorno de pruebas cuidadosamente elaborado con el fin de exponer las propiedades de interés que se buscan verificar sobre el mismo. Una de las técnicas de testing de software que ha captado más atención en los últimos años tanto en la industria como en la academia es la que se conoce como *random testing*[20]. La misma consiste en "alimentar" las entradas de los programas a verificar usando datos generados aleatoriamente en búsqueda de aquellos que resultan inesperados o que son manipulados de manera incorrecta por el sistema bajo pruebas y que, en un caso límite, podrían representar una amenaza a la seguridad del mismo[15]. Esta técnica se ha mostrado capaz de encontrar vulnerabilidades previamente desconocidas en una multitud de sistemas de software de manera relativamente rápida y, principalmente, a un costo operativo muy bajo[1, 22].

La alta complejidad de los sistemas de software modernos fuerza a que el proceso de testing de los mismos sea tan automatizable como sea posible[16], buscando lograr escalabilidad respecto de la velocidad de expansión de los mismos. Es por esto que actualmente existen mecanismos que permiten derivar herramientas de testing aleatorio de manera automática basándose

en el código fuente del sistema a verificar[12, 18, 30], evitando de esa manera la necesidad de programar una suite de pruebas específicas para cada nuevo desarrollo de software. Sin embargo, la efectividad del proceso de testing al utilizar estos mecanismos de derivación automática suele resultar deficiente si se la compara con aquella conseguida por suites de prueba construidas específicamente para el sistema a verificar. Esto se debe en parte a la dificultad inherente de extraer los invariantes de cualquier sistema de software a partir únicamente de su código fuente. Luego, esta incapacidad de reconocer las propiedades del sistema a verificar resulta en una capacidad limitada de generar valores aleatorios capaces de poner a prueba a la implementación del mismo respecto del comportamiento esperado.

En este trabajo se considera el escenario donde el desarrollador del sistema no está completamente al tanto de las propiedades e invariantes que los datos de entrada del mismo deben cumplir. Esto constituye una hipótesis válida para técnicas de testing como lo son el *testing de penetración*, donde el encargado de verificar el sistema usualmente complementa la generación aleatoria de datos con su posterior mutación mediante el uso de *fuzzers*[7, 17, 27, 28, 33] en búsqueda de conseguir que los sistemas fallen irrecuperablemente o aborten—un tipo de anomalía que puede acarrear amenazas graves a la seguridad del mismo.

Una de las mayores contribuciones de este trabajo es la de proveer fundamentos matemáticos que permitan modelar el proceso de generación de datos aleatorios. En particular, buscamos calcular analíticamente la distribución de los datos generados mediante generadores de datos derivados automáticamente, usando para ésto un enfoque basado en tipos de datos algebraicos en Haskell. En este trabajo utilizamos un modelo estocástico relativamente simple conocido como *proceso de ramificación*[19], el cual fue concebido para estudiar la evolución y posible extinción de las familias reales en Europa. Este modelo se ha extrapolado con éxito a diversas áreas de la ciencia, y en esta tesina mostramos como puede ser utilizado para modelar la generación aleatoria de datos de manera sencilla. Luego, a partir de este modelo de generación aleatoria, diseñamos una heurística capaz de optimizar de manera automática los parámetros de generación con el fin de satisfacer ciertas propiedades en la distribución de datos conseguida.

Finalmente, proveemos una implementación en Haskell de nuestras ideas y las evaluamos generando entradas aleatorias para una serie de programas de envergadura considerable. En nuestros experimentos, los resultados muestran que este proceso de optimización de paráme-

tros de generación produce una mejora considerable en términos de cubrimiento de código fuente respecto de las herramientas del estado del arte para realizar random testing automático en Haskell.

El resto de esta tesina se organiza como sigue: el Capítulo 2 detalla algunos conceptos preliminares. En el Capítulo 3 se presenta el estado del arte en materia de random testing automático en Haskell junto con sus limitaciones. El Capítulo 4 introduce un paralelismo entre la generación de valores aleatorios basada en la definición de su tipo de datos y los procesos de ramificación. Este paralelismo luego es extendido para soportar tipos de datos más expresivos soportados por el sistema de tipos de Haskell. El Capítulo 5 presenta la implementación de una herramienta * capaz de derivar generadores de datos aleatorios de manera automática, y cuya distribución de generación de datos es optimizada utilizando la teoría previamente formulada. Junto a esto, este capítulo presenta la evaluación empírica de casos de estudio que ponen a prueba el poder de los generadores de datos aleatorios derivados por la herramienta antes presentada ante sistemas de software de notable complejidad. Las conclusiones de esta tesina y el trabajo futuro se presentan en el Capítulo 6.

*Disponible en `https://bitbucket.com/agustinmista/dragen`.

2

Preliminares

Este capítulo sirve el propósito de introducir al lector ante algunos conceptos relevantes para el desarrollo de esta tesina. En particular, se detallan nociones básicas de Haskell, el lenguaje de programación utilizado en el presente trabajo; junto a *Template Haskell* y *QuickCheck*, la extensión de Haskell que le añade al mismo capacidades de meta-programación y el conocido framework de random testing, respectivamente.

2.1. Haskell

Haskell[26] es un lenguaje de programación *funcional puro*, *fuertemente tipado estáticamente* y de *evaluación lazy*. Es decir, un lenguaje de programación con: i) una clara separación entre funciones *puras*, sin efectos secundarios, y computaciones *impuras* que producen efectos como entrada/salida, manipulación de referencias mutables, acceso a un estado global, etc. ii) un sistema de tipos que conoce *en tiempo de compilación* el tipo de cada expresión, y que es suficientemente sofisticado como para atrapar errores comunes de programación en etapas tempranas de desarrollo, y iii) que sólo ejecuta funciones y calcula resultados cuando es realmente necesario, permitiendo expresar estructuras de datos infinitas o utilizar programación

circular, entre otros. Estos aspectos dan lugar a un gran número de posibles optimizaciones sobre el código fuente, junto a la posibilidad de expresar abstracciones de alto nivel difícilmente representables en otros lenguajes de programación convencionales.

Haskell posee un sistema de tipos muy expresivo que le permite al usuario definir nuevos tipos de datos (o simplemente tipos) de manera muy concisa utilizando un enfoque *algebraico*. Es decir, como sumas (o uniones disjuntas) y productos (o pares) de otros tipos de datos o de sí mismo. Por ejemplo, el tipo de los valores booleanos se puede definir como:

$$\texttt{data}\ \mathit{Bool} = \mathit{True} \mid \mathit{False}$$

Donde `data` es una palabra clave reservada que indica que estamos definiendo un nuevo tipo de datos. En este caso, *Bool* es representado como la unión disjunta (denotada por el símbolo |) de los *constructores de tipo* (o simplemente *constructores*) *True* y *False*, los cuales son los *únicos* habitantes del tipo *Bool*, y por ende la única manera de construir un nuevo valor de este tipo es a través de los mismos. Luego, mediante *pattern matching* podemos definir fácilmente funciones que manipulen valores de este tipo de datos:

$$\begin{aligned}&\mathit{not} :: \mathit{Bool} \to \mathit{Bool}\\&\mathit{not}\ \mathit{True} = \mathit{False}\\&\mathit{not}\ \mathit{False} = \mathit{True}\end{aligned}$$

$$\begin{aligned}&\mathit{or} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\\&\mathit{or}\ \mathit{True}\ _ = \mathit{True}\\&\mathit{or}\ \mathit{False}\ p\ = p\end{aligned}$$

En particular, *not* está definida como una función que toma un booleano y devuelve otro booleano, mientras que *or* es una función que toma dos booleanos y devuelve otro booleano. En las definiciones anteriores, el símbolo :: denota una *signatura de tipo* e indica el tipo de una expresión, mientras que el símbolo (_) representa un patrón *comodín* o *wildcard*, el cual evita la necesidad de darle un nombre a aquellos patrones que luego simplemente descartamos.

Por otro lado, es posible definir el producto de dos tipos de datos mediante constructores de tipos con múltiples argumentos. Así, el tipo de los pares de números enteros puede definirse usando un constructor con dos argumentos de tipo *Int* como se muestra a continuación:

```
data Pair = Pair Int Int
```

De esta manera, un valor de tipo *Pair* almacena dos valores enteros, por ejemplo (*Pair* 4 2).

Además, cada tipo de datos puede estar parametrizado por otros tipos de datos. Esto hace posible definir contenedores de datos genéricos parametrizados por los tipos de los valores que almacenan. Por ejemplo, el tipo de los pares de enteros puede generalizarse a pares de cualquier par de tipos *a* y *b* como se muestra a continuación:

```
data Pair a b = Pair a b
```

Donde *a* y *b* pueden ser luego instanciados con cualquier otro tipo de datos. Por ejemplo:

```
p :: Pair Int String
p = Pair 42 "Hola"
```

Adicionalmente, cada tipo de datos puede estar definido de manera recursiva, haciendo referencia a sí mismo en los argumentos de sus constructores. De esta manera, el tipo de los árboles binarios que almacenan información genérica de tipo *a* en los nodos puede representarse como:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Y un posible árbol binario con información entera puede representarse como:

```
t :: Tree Int
t = Node Leaf 42 (Node Leaf 24 Leaf)
```

Dado que la manera natural de definir tipos de datos que hacen referencia a sí mismos en Haskell es la recursión, este mecanismo es también la manera natural de definir funciones que manipulan valores de estos tipos de datos. Por ejemplo, podemos definir una función que calcula la profundidad de un *Tree a* como sigue:

$$
\begin{array}{l}
\mathit{depth} :: \mathit{Tree}\ a \to \mathit{Int} \\
\mathit{depth}\ \mathit{Leaf} = 0 \\
\mathit{depth}\ (\mathit{Node}\ l\ _\ r) = 1 + \mathit{max}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r)
\end{array}
$$

Notar como esta función calcula la profundidad de un árbol binario en base a la profundidad de sus sub-árboles, la cual es calculada recursivamente en el caso de recibir como entrada un árbol no vacío (caso $(\mathit{Node}\ \cdots)$). Merece la pena remarcar que la signatura de la función *depth* utiliza la *variable de tipo a*, que representa un tipo genérico el cual no influye en la implementación de la misma—el pattern matching sobre el constructor *Node* simplemente descarta la información contenida en los nodos, ya que la misma no es necesaria a la hora de calcular la profundidad de un árbol binario. De esta manera, *depth* puede ser utilizada con cualquier instancia concreta de *Tree a*.

Finalmente, es muy común que un usuario desee poder utilizar funciones como la suma $(+)$ o la igualdad $(\equiv)$ de valores de manera uniforme dentro de su código fuente. Para esto Haskell provee un mecanismo de sobrecarga de funciones conocido como *polimorfismo ad-hoc*, que permite definir interfaces genéricas llamadas *clases de tipos*[36], las cuales comprenden una serie de signaturas de tipos que pueden ser luego *instanciadas* de manera concreta para cada tipo de datos que se desee adherir a dicha clase. Por ejemplo, Haskell define la clase *Eq* de aquellos tipos de datos que pueden ser comparados por igualdad:

$$
\begin{array}{l}
\texttt{class}\ \mathit{Eq}\ a\ \texttt{where} \\
\quad (\equiv) :: a \to a \to \mathit{Bool}
\end{array}
$$

En la definición anterior, *Eq* especifica una interfaz formada únicamente por la función $(\equiv)$, la cual nos permite evaluar la igualdad de dos valores del mismo tipo genérico *a*. Luego, si desea-

mos sobrecargar la función $(\equiv)$ para que pueda ser utilizado para comparar valores booleanos debemos proveer una instancia concreta de *Bool* para esta clase de tipos:

```
instance Eq Bool where
  True  ≡ True  = True
  False ≡ False = True
  _     ≡ _     = False
```

Sin embargo, si deseamos definir una instancia concreta de la clase de tipos *Eq* para nuestros árboles binarios previamente definidos, además debemos considerar que el tipo de datos que estos almacenan también debe poder ser comparado por igualdad. Esta condición se logra imponiendo una restricción o *constraint* en la definición de la instancia de clase:

```
instance Eq a ⇒ Eq (Tree a) where
  Leaf         ≡ Leaf         = True
  Node l₁ x₁ r₁ ≡ Node l₂ x₂ r₂ = (x₁ ≡ x₂) ∧ (l₁ ≡ l₂) ∧ (r₁ ≡ r₂)
```

Esta restricción nos permite usar $(\equiv)$ para evaluar la igualdad de los valores almacenados por los nodos dentro la definición de la instancia. Una vez definida esta instancia, Haskell limita al programador a utilizar $(\equiv)$ sólo con aquellos valores de *Tree a* en los que el tipo de datos *a* también puede ser comparado mediante $(\equiv)$.

Una clase de tipos muy notable en Haskell es la correspondiente a los tipos de datos *monádicos*[29], la cual es utilizada para estructurar varios tipos de efectos computacionales tales como manejo de errores, entrada/salida, manipulación de estado global, etc. Su interfaz está definida en la clase de tipos *Monad*, cuya definición abreviada se muestra a continuación:

```
class Monad m where
  return :: a → m a
  (≫=)  :: m a → (a → m a) → m a
```

Donde *return* "inyecta" un valor puro de tipo a dentro de un contexto computacional de tipo $m\ a$ sin producir ningún efecto, mientras que $(\gg=)$, pronunciado como *bind*, se encarga de componer secuencialmente dos computaciones monádicas en un nueva computación monádica, utilizando posiblemente la información calculada en cada paso para controlar el flujo de datos en los pasos posteriores, y produciendo posiblemente algún efecto computacional en el proceso. Esta clase es tan utilizada en la práctica, que Haskell de hecho provee azúcar sintáctico para aliviar el trabajo a la hora de programar, conocida como *notación-*do:

$$
\begin{array}{l}
myFun = \mathsf{do} \\
\quad x \leftarrow f \\
\quad y \leftarrow g \\
\quad return\ (h\ x\ y)
\end{array}
$$

Notación do

$$
\begin{array}{l}
myFun = \\
\quad f \gg= (\lambda x \rightarrow \\
\quad\quad g \gg= (\lambda y \rightarrow \\
\quad\quad\quad return\ (h\ x\ y)))
\end{array}
$$

Versión desazucarada

Donde la expresión $(\lambda x \rightarrow y)$ denota una λ-función o *función anónima* que recibe el parámetro x y retorna la expresión y.

Hasta aquí hemos presentado algunos conceptos básicos referidos al lenguaje de programación Haskell y sus construcciones sintácticas. A continuación presentamos una extensión del compilador de Haskell que permite tratar al código fuente como datos, manipulando o generando nuevo código fuente en tiempo de compilación.

2.2. Template Haskell

Template Haskell [32] es una extensión de Haskell que proporciona herramientas de metaprogramación en tiempo de compilación. Intuitivamente, *Template Haskell* provee al lenguaje nuevas construcciones que le permiten al usuario manipular valores desde y hacia sintaxis concreta. Esto es, convertir valores Haskell en código fuente y viceversa. Junto a esto, *Template Haskell* provee una representación de los árboles de sintaxis abstracta (AST) de Haskell construida dentro del mismo lenguaje, lo que permite tratar al código fuente como datos normales. Todo esto permite tanto hacer *introspección* del código fuente para obtener información relevante del entorno actual, como también generar nuevo código fuente que es luego compilado

junto al resto del programa, preservando la seguridad de tipos del lenguaje. *Template Haskell* posee una gran cantidad de usos prácticos y se ha vuelto muy popular en los últimos años, llegando a ser el mecanismo de meta-programación de preferencia de muchas librerías de uso masivo como *aeson*[6] y *lens*[13].

La generación programática de código fuente requiere que ciertas capacidades de programación estén disponibles para el usuario, tales como: I) la habilidad de generar nombres únicos que no puedan ser capturados, II) la capacidad de recuperar información acerca de un tipo de datos, función, módulo, etc. mediante su nombre, III) la habilidad de manipular un estado interno que es compartido por todo el código auto-generado dentro del mismo módulo, y IV) la posibilidad de efectuar entrada/salida durante la compilación. Dado que, como se mostró antes, estas capacidades suelen conseguirse en Haskell mediante el uso de tipos de datos monádicos, *Template Haskell* define el tipo monádico Q que permite estructurar estos efectos de manera controlada. Así, un valor de tipo $Q\ a$ puede verse como una computación monádica que puede generar nuevo código fuente de tipo a en tiempo de compilación, donde a es uno de los siguiente tipos de datos:

- *Dec*, que representa declaraciones de funciones y tipos de datos en el top-level.
- *Exp*, representando expresiones como $(x + 1)$ ó $(\lambda x \to x * 2)$.
- *Type*, que representa expresiones de tipos como *Int* o *Maybe*.
- *Pat*, útil para definir patrones usados al hacer pattern matching, como $(Tree\ l\ _\ r)$.

Con toda esta maquinaria a disposición, el programador puede definir meta-funciones que sinteticen código fuente en demanda de manera relativamente sencilla. A continuación veremos un ejemplo clásico de un caso de uso de esta herramienta.

Imaginemos que nuestro código fuente requiere manipular grandes tuplas de datos:

$$bigTuple :: (T_1, T_2, \cdots, T_k)$$

Haskell únicamente provee proyecciones $fst :: (a, b) \to a$ y $snd :: (a, b) \to b$ para tuplas de exactamente dos elementos. Esto fuerza al programador a definir manualmente todas las

proyecciones necesarias para las tuplas de datos que su código fuente debe manipular, lo cual resulta un trabajo tedioso y repetitivo. Afortunadamente, usando *Template Haskell* puede definir una familia de proyecciones de tuplas de datos en tiempo de compilación. En primer lugar, se debe activar *Template Haskell* agregando el siguiente *pragma de compilación* en la cabecera del módulo Haskell:

```
{-# LANGUAGE TemplateHaskell #-}
```

Luego, se puede definir una meta-función *proj*[*] que sintetice la función proyección necesaria, indexada por el tamaño de la tupla de datos, y el índice de la misma que se quiere proyectar:

```
proj :: Int -> Int -> Q Exp
proj tupSize index = do
  x <- newName "x"
  lam1E
    (tupP
      (replicate index wildP                         -- Primeros índices
       ++ [varP x]                                   -- Índice deseado
       ++ replicate (tupSize - index - 1) wildP))    -- Resto de los índices
    (varE x)
```

En el ejemplo anterior, *proj* genera un nuevo nombre fresco de variable, prefijado por "x", que es luego utilizado al generar una nueva función anónima (ver *lam1E*). La misma recibe una tupla de datos del tamaño especificado (ver *tupP*), para la cual se hace pattern matching únicamente sobre el elemento en el índice deseado (ver *varP x*), retornando el mismo (ver *varE x*) y descartando el resto de los elementos de la tupla mediante un wildcard (ver *wildP*).

Luego, el desarrollador puede llamar a *proj* interpolándola dentro de su código fuente mediante la construcción sintáctica $(), la cual será reemplazada en tiempo de compilación por una función de proyección de tuplas sintetizada por *Template Haskell*. Si se activa la bande-

[*] Este ejemplo puede encontrarse en el código fuente de la librería *TupleTH*.

ra de compilación `-ddump-splices`, es posible observar el código generado por esta metafunción. Por ejemplo:

$$f :: Int$$
$$f = \$(proj\ 6\ 2)\ (1,1,2,3,5,8)$$
$$\Longrightarrow$$
$$f = (\lambda(_,_,x_0,_,_,_) \rightarrow x_0)\ (1,1,2,3,5,8)$$

Donde el símbolo $\Longrightarrow$ divide el código fuente escrito por el programador del código fuente generado por *Template Haskell*.

En este trabajo se hace uso intensivo de *Template Haskell* como mecanismo de derivación de generadores de datos aleatorios en tiempo de compilación. A continuación se presenta el framework por excelencia para llevar a cabo testing aleatorio en Haskell.

2.3. QuickCheck

QuickCheck [9] es un framework de testing aleatorio implementado en Haskell como un mero lenguaje embebido de dominio específico[21], el cual adopta un modelo de testing aleatorio basado en propiedades o *Random Property-based Testing* (RPBT) en Inglés. Este modelo consta esencialmente de dos conceptos fundamentales: i) las propiedades que se desea que el sistema a verificar cumpla, y ii) los generadores de datos aleatorios para los aquellos tipos de datos manipulados por el sistema. Luego, el proceso de testing se resume a generar valores aleatorios de los tipos involucrados en las propiedades deseadas para el sistema, en búsqueda de contraejemplos capaces de vulnerar alguna/s de ellas.

Es muy importante destacar que, si bien esta metodología de testing es muy efectiva en la práctica, la misma es capaz de descubrir errores de programación o de diseño de un sistema pero *no de garantizar su ausencia*. Es por esto que se suelen usar ciertos criterios como lo es el grado de cubrimiento del código desencadenado por el proceso de testing como medida de la efectividad y confiabilidad del mismo[8].

QuickCheck provee herramientas para modelar ambos conceptos de manera muy sucinta. En el caso de las propiedades, las mismas son definidas utilizando predicados lógicos que cuantifican sobre los tipos de datos manipulados por el sistema a verificar y que son relevantes para la propiedad en cuestión. Por otro lado, los generadores de datos aleatorios se definen utilizando una serie de primitivas que esta herramienta provee para modelar el proceso de generación aleatoria de datos. A continuación se muestra un ejemplo del flujo de trabajo usual requerido para validar una componente de software mediante *QuickCheck*.

Supongamos que se está implementando una librería que permite almacenar datos eficientemente mediante árboles binarios. Con este fin, el programador decide reutilizar el tipo de datos *Tree* previamente definido:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Dado que este tipo de datos es usado para almacenar datos que deben ser recuperados con el menor costo computacional posible, una propiedad altamente deseable es que los mismos puedan mantenerse balanceados en todo momento. El programador define entonces una función *balance* que se encarga de tomar un árbol binario (posiblemente) desbalanceado y retornar un árbol binario balanceado:

```
balance :: Tree a → Tree a
```

Luego, se quiere verificar mediante *QuickCheck* que la función *balance* se comporta como es esperado. La primer pieza del rompecabezas consiste en definir una propiedad—a la que llamamos *prop_balance*—que valide si efectivamente *balance* retorna árboles binarios balanceados:

```
prop_balance :: Tree a → Bool
prop_balance t = isBalanced (balance t)
```

$$
\begin{aligned}
&\mathit{isBalanced} :: \mathit{Tree}\ a \to \mathit{Bool}\\
&\mathit{isBalanced}\ \mathit{Leaf} \qquad\quad = \mathit{True}\\
&\mathit{isBalanced}\ (\mathit{Node}\ l\ _\ r) = \mathit{abs}\ (\mathit{depth}\ l - \mathit{depth}\ r) \leqslant 1\\
&\qquad\qquad\qquad\qquad\quad \wedge \mathit{isBalanced}\ l\\
&\qquad\qquad\qquad\qquad\quad \wedge \mathit{isBalanced}\ r
\end{aligned}
$$

La propiedad anterior define el comportamiento esperado para la función *balance*. Dado que existen múltiples nociones de balance de árboles, en este ejemplo particular definimos arbitrariamente que un árbol binario se encuentra balanceado si sus sub-árboles izquierdo y derecho poseen una profundidad que difiere en a lo sumo 1, y que a su vez los mismos se encuentran balanceados. Cabe destacar que, en un escenario real, *prop_balance* seguramente no cubre todos los detalles de implementación de un algoritmo de balanceo de árboles binarios, por lo que seguramente harán falta definir más propiedades a verificar sobre *balance*. Sin embargo, *prop_balance* es suficiente para ilustrar el punto en cuestión.

Una vez especificado el comportamiento esperado del sistema mediante propiedades, es necesario proveer una manera de generar valores aleatorios de los tipos involucrados en la definición de las mismas. Con este fin, *QuickCheck* define la clase de tipos *Arbitrary*, de los tipos de datos que pueden ser generados aleatoriamente:

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

La función sobrecargada *arbitrary* representa un generador de datos aleatorios del tipo monádico *Gen a*, donde el tipo concreto generado *a* es resuelto en tiempo de compilación mediante inferencia de tipos. *QuickCheck* provee instancias de esta clase de tipos para los tipos primitivos más comunes como *Int*, *Bool*, *Double*, etc. Sin embargo, el programador debe proveer generadores de datos aleatorios para todos los tipos de datos definidos por él. La manera más usual de proveer dichos generadores es proporcionando una instancia de la clase de tipos *Arbitrary* para cada tipo de datos en concreto. En nuestro caso particular, definiendo una instancia concreta de *Arbitrary* (*Tree a*), por ejemplo como se muestra a continuación.

```
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary = oneof [genLeaf, genNode]
    where
      genLeaf = return Leaf
      genNode = do
        l ← arbitrary
        r ← arbitrary
        x ← arbitrary
        return (Node l x r)
```

Dado que nuestros árboles binarios almacenan información genérica, para definir una instancia de *Arbitrary* apropiada además debemos requerir en que el tipo de datos de la información almacenada por los mismos pueda ser generado aleatoriamente, utilizando una restricción en la signatura de la instancia definida como se mostró anteriormente en este capítulo. En la definición anterior , el proceso de generación aleatoria de árboles binarios está dictado por la función primitiva de *QuickCheck* $oneof :: [Gen\ a] \rightarrow Gen\ a$, que devuelve un generador aleatorio eligiendo al azar y con la misma probabilidad de entre una lista de posibles generadores aleatorios, cada uno de ellos especializado en la generación de un tipo de árboles binario en particular. De esta manera, el proceso de generación aleatoria puede elegir entre generar una hoja (caso *genLeaf*) o generar un nodo (caso *genNode*). En el caso particular en que un nodo es generado, además se deben generar recursivamente dos sub-árboles y un valor de tipo *a* llamando a *arbitrary*, con el fin de "llenar" los argumentos de este constructor.

Una vez definido nuestro generador aleatorio de árboles binarios, podemos hacer que *QuickCheck* genere valores aleatorios de manera sencilla:

```
ghci> generate (arbitrary :: Gen (Tree Int))
Node (Node Leaf 3 (Node Leaf (-6) Leaf)) 13 (Node Leaf 42 Leaf)
ghci> generate (arbitrary :: Gen (Tree Bool))
Node Leaf True (Node Leaf False (Node Leaf False Leaf))
```

Finalmente, el programador puede verificar que *balance* se comporta como es esperado lanzando una *campaña de testing*. Durante la misma, *QuickCheck* se encargará de generar aleatoriamente valores del tipo de los argumentos de entrada de las propiedades definidas, intentando encontrar un conjunto de valores aleatorios que falsifiquen alguna de ellas. Mas precisamente, en nuestro caso *QuickCheck* generará árboles binarios en búsqueda de algún valor t tal que $prop_balance\ t \equiv False$. Si se da el caso que ningún valor generado aleatoriamente es capaz de falsificar nuestra propiedad, *QuickCheck* entonces mostrará un mensaje de éxito:

```
ghci> quickCheck prop_balance
+++ OK, passed 100 tests.
```

Si en cambio se genera aleatoriamente un valor capaz hacer que *balance* funcione de manera incorrecta según nuestra propiedad, el mismo es *minimizado* (i.e. reducido estructuralmente) y ofrecido al usuario como contraejemplo para que éste efectúe los análisis pertinentes. A continuación se muestra un posible contraejemplo para nuestra propiedad.

```
ghci> quickCheck prop_balance
*** Failed! Falsifiable (after 25 tests and 14 shrinks):
(Node (Node (Node Leaf () Leaf) Leaf) Leaf)
```

Cabe destacar que, en los ejemplos anteriores *QuickCheck* se encarga automáticamente de instanciar el tipo de datos *a* que parametriza nuestros árboles binarios con un tipo de datos concreto apropiado. [**] De esta manera se evita la necesidad de utilizar tipos de datos concretos en aquellas propiedades que expresen características estructurales del tipo de datos en cuestión—el algoritmo de balanceo de un árbol binario no depende en ninguna manera de la información que éstos almacenan.

Si bien la tarea de definir propiedades y generadores de datos aleatorios puede parecer sencilla, la misma se vuelve tediosa y propensa a errores en sistemas de envergadura no trivial. Actualmente existen soluciones que permiten subsanar parcialmente esta problemática tanto por el lado de las propiedades[3, 11], como por el lado de los generadores de datos aleatorios[12, 17, 30], y que contribuyen a lo que se suele denominar como *Automatic Random Testing*.

[**] En particular, utiliza el tipo unitario de Haskell (), el cual contiene un único habitante ().

En este trabajo nos concentramos específicamente en el proceso automático de obtención de generadores de datos aleatorios a partir de las definiciones de sus tipos de datos. El siguiente capítulo presenta el estado del arte en materia de derivación automática de generadores de datos aleatorios en Haskell.

"It is the basic condition of life, to be required to violate your own identity."

Philip K. Dick.

3

Random Testing Automático en Haskell

En este capítulo se analiza el estado del arte relacionado al random testing automático en Haskell. En particular, se presentan tres herramientas de derivación automática de generadores de datos aleatorios disponibles en la actualidad, junto con un análisis de sus principales desventajas, las cuales dan lugar a la motivación de este trabajo.

3.1. *DERIVE*

La manera más simple de obtener un generador de datos aleatorios para un tipo de datos dado de manera automática es la implementada por la librería de Haskell *derive* [30]. Si suponemos que se desea obtener un generador aleatorio para el siguiente tipo de datos:

```
data T = A | B T T | C T T
```

Usando *derive* podemos conseguirlo fácilmente agregando la siguiente línea de código en el top-level de algún módulo donde T sea alcanzable:

```
$(derive makeArbitrary ''T)
 ⟹
instance Arbitrary T where
  arbitrary = do
    x ← choose (0, 2)
    case x of
      0 → return A
      1 → do
        b1 ← arbitrary
        b2 ← arbitrary
        return (B b1 b2)
      2 → do
        c1 ← arbitrary
        c2 ← arbitrary
        return (C c1 c2)
```

El símbolo $''(_) :: \mathit{Name}$ se conoce como el operador de *promoción de tipos*, y nos permite obtener el nombre interno de un tipo de datos. Así, $''T :: \mathit{Name}$ representa el nombre interno del tipo T.

Los generadores aleatorios sintetizados usando *derive* utilizan un enfoque semánticamente equivalente al visto en el capítulo anterior. En lugar de utilizar la operación primitiva de *QuickCheck oneof*, este generador simula el mismo comportamiento generando aleatoriamente, y de manera uniforme, un número entero entre 0 y 2 mediante la función primitiva de *QuickCheck choose* $:: (\mathit{Int}, \mathit{Int}) \to \mathit{Gen}\ \mathit{Int}$, el cual es luego utilizado para decidir qué constructor generar mediante una instrucción `case`. Luego, en cada caso se genera el constructor elegido anteriormente al azar, generando recursivamente los sub-términos que sean necesarios.

Si bien esta librería ofrece una mejora considerable al proceso de testing automático, la implementación de la misma posee un serio inconveniente: la misma no contempla ni controla la posibilidad de que ocurran bucles infinitos en la generación aleatoria de valores de tipos de datos recursivos. Ante la presencia de un constructor recursivo, los generadores sintetizados

mediante *derive* simplemente generan sub-términos llamándose a si mismos de manera recursiva, con la misma probabilidad de elegir nuevamente un constructor recursivo en el siguiente paso una y otra vez. Este comportamiento empeora a medida que se aumenta tanto el número de constructores recursivos como el número de argumentos recursivos en cada uno de ellos, dado que esto aumenta la probabilidad de escoger un constructor recursivo cada vez que se necesita generar aleatoriamente un sub-término. Para ilustrar la magnitud de este problema, la siguiente figura muestra la distribución de datos en la generación aleatoria de 100.000 valores de tipo T utilizando el generador de datos obtenido mediante *derive*, en función de cantidad de constructores presentes en cada valor generado. La barra sólida a la derecha indica aquellos valores que produjeron bucles infinitos durante su generación. *

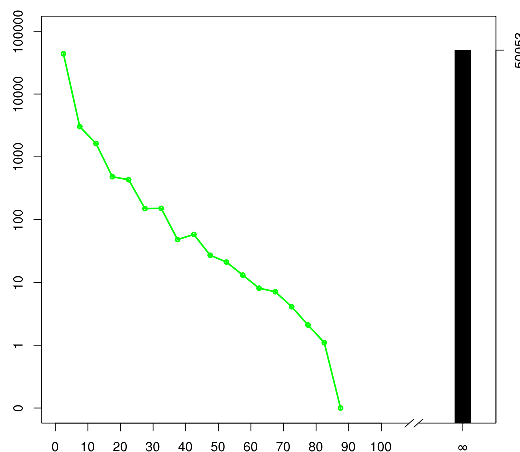


Figura 3.1: Distribución de valores obtenida utilizando el generador obtenido con *derive* para el tipo de datos T.

En la figura anterior puede observarse cómo la generación de valores se ve atascada en un bucle infinito prácticamente la mitad de las veces que se genera un nuevo valor aleatorio. Este comportamiento puede ser formalizado utilizando el concepto de *función generadora de probabilidad* de una variable aleatoria discreta X, la cual representa la cantidad de valores recursivos generados aleatoriamente en cada paso del proceso de generación. Formalmente, se sabe que la probabilidad de extinción d de un valor generado aleatoriamente (y por ende la finitud del proceso aleatorio que lo genera) puede ser calculada encontrando el menor punto

*Los bucles infinitos en la generación de valores fueron identificados mediante el uso de un timeout suficientemente grande.

fijo de la función generadora de probabilidad correspondiente a dicho proceso, la cual viene dada por:

$$G(z) = E(z^X) = \sum_{k=0}^{\infty} P(X = k) \cdot z^k$$

Si aplicamos esta ecuación a nuestro tipo de datos T debemos considerar que en cada paso sólo es posible generar cero o dos sub-términos, mediante el constructor A y los constructores B y C, respectivamente. De este modo resulta que $P(X = k) = 0$ para todo $k \notin \{0, 2\}$. Luego, podemos calcular la probabilidad de extinción de un valor de tipo T generado aleatoriamente, donde se considera que cada constructor tiene la misma probabilidad de ser generado simplemente encontrando la menor solución a la siguiente ecuación:

$$\begin{aligned}
d &= G(d) = P(X = 0) + P(X = 2) \cdot d^2 \\
d &= p_A + (p_B + p_C) \cdot d^2 \\
d &= (1/3) + (2/3) \cdot d^2 \\
d &= 1/2
\end{aligned}$$

De esta manera podemos confirmar que se espera que la generación de valores usando este generador aleatorio sólo termine en promedio la mitad de las veces. Este comportamiento es altamente indeseable, ya que no resulta factible manipular valores infinitos de manera práctica durante el proceso de testing de software, junto con el hecho de que la mayoría del software que se desarrolla espera entradas finitas. Aquellos tipos de software que aceptan entradas infinitas, como lo son los sistemas operativos o las implementaciones de protocolos de comunicación podrían verse beneficiados por un paradigma de testing aleatorio que permita generar valores infinitos de manera controlada. Sin embargo, esta clase de sistemas constituye un objeto de estudio en sí mismo, y usualmente requiere de una maquinaria especializada para su validación.

Dada la presencia de bucles infinitos usando este tipo de generadores aleatorios, en este trabajo omitimos realizar cualquier tipo de comparación entre *derive* y el resto de los mecanismos de derivación automática de generadores aleatorios abordados por el mismo.

El segundo mecanismo de derivación automática de generadores aleatorios en Haskell que discutimos en este capítulo es *MegaDeTH* [17, 18] (por *Mega Derivation Template Haskell*). Esta herramienta nace como una alternativa a *derive* que soluciona sus principales inconvenientes. En primer lugar, *MegaDeTH* permite derivar automáticamente generadores de datos para el tipo de datos especificado y todos sus tipos anidados—usando *derive* el programador debe encargarse de ejecutar *makeArbitrary* explícitamente para cada tipo involucrado transitivamente en la definición del tipo especificado en primer lugar. Esto se logra en la práctica calculando un *sort topológico* de dependencias de tipos, con el fin de derivar todos los generadores de datos aleatorios necesarios antes de derivar el generador de datos para el tipo deseado.

Más importante aún, *MegaDeTH* es suficientemente sofisticado como para derivar generadores de datos aleatorios que sólo producen valores estrictamente finitos, haciendo uso de mecanismos de generación acotada provistos por *QuickCheck*, los cuales permiten controlar el número máximo de llamadas recursivas que cada generador de datos puede efectuar. Para ilustrar sencillamente este último punto haremos uso de el siguiente tipo de datos, el cual representa árboles binarios que no almacenan información, y con tres clases distintas de hojas:

$$\texttt{data}\ Tree_{3L} = Leaf_A \mid Leaf_B \mid Leaf_C \mid Node\ Tree_{3L}\ Tree_{3L}$$

Los generadores aleatorios sintetizados utilizando *MegaDeTH* hacen uso de la función primitiva de *QuickCheck* $sized :: (Int \to Gen\ a) \to Gen\ a$ que permite construir un generador aleatorio a partir de una función (de tipo $Int \to Gen\ a$) que limita el número posible de llamadas recursivas que pueden efectuarse durante el proceso de generación. El valor entero que recibe esta función es conocido como el *tamaño de generación*, y es un parámetro interno del proceso de generación el cual se enlaza implícitamente en cada llamada sucesiva a *arbitrary* mediante un efecto monádico. De esta manera, el tamaño de generación puede utilizarse como parámetro de control para modificar el comportamiento de la generación aleatoria luego de cada llamada recursiva a *arbitrary*. A continuación se muestra una versión simplificada, pero semánticamente equivalente del generador de datos derivado por *MegaDeTH* para el

tipo de datos $Tree_{3L}$:

```
$(devArbitrary ''Tree3L)
 ==>
instance Arbitrary Tree3L where
  arbitrary = sized gen
    where
      gen 0 = genTerm
      gen n = genAny n
      genTerm = oneof [return LeafA, return LeafB, return LeafC]
      genAny n = oneof [genTerm, genNonTerm n]
      genNonTerm n = do
        t1 <- resize (n `div` 2) arbitrary
        t2 <- resize (n `div` 2) arbitrary
        return (Node t1 t2)
```

Figura 3.2: Generador de datos aleatorios derivado automáticamente por *MegaDeTH* para el tipo de datos $Tree_{3L}$.

Puede notarse como este generador aleatorio utiliza *sized* para controlar el comportamiento de una función auxiliar $gen :: Int \rightarrow Gen\ Tree_{3L}$, la cual se encarga de gobernar el proceso aleatorio en función del tamaño de generación recibido implicítamente. Si el tamaño de generación es 0 (caso $gen\ 0$), el proceso aleatorio sólo puede generar constructores terminales (ver *genTerm*), finalizando así el proceso de generación. Si en cambio el tamaño de generación es positivo (caso $gen\ n$), el proceso de generación elige aleatoriamente entre generar un constructor terminal o un constructor no-terminal (ver $genAny\ n$). A su vez, en el caso de que se elige aleatoriamente generar un constructor no-terminal (ver $genNonTerm\ n$), cada llamada recursiva a *arbitrary* se efectúa reduciendo el tamaño de generación por un factor que depende de la cantidad de argumentos recursivos que el constructor elegido posee. Esto se consigue gracias a la función de *QuickCheck* $resize :: Int \rightarrow Gen\ a \rightarrow Gen\ a$ que se encarga de modificar el tamaño de generación implícito para el generador aleatorio que recibe como parámetro (ver $resize\ (n\ `div`\ 2)\ arbitrary$). De esta manera, *MegaDeTH* asegura que todo valor generado aleatoriamente es finito, puesto que toda generación recursiva de sub-términos

reduce progresivamente el tamaño de generación hasta llegar a 0, donde la generación escoge un constructor terminal y el proceso de generación se detiene.

MegaDeTH se ha mostrado de gran utilidad en la práctica, y es utilizado intensivamente por *QuickFuzz* [17, 18], una herramienta de testing automático que combina testing aleatorio y fuzzing para descubrir fallos en sistemas de software de naturaleza sumamente variada, haciendo uso de librerías de datos previamente existentes en el ecosistema de Haskell y que codifican mediante tipos de datos la estructura de muchos formatos de archivos de uso masivo. Sin embargo, *MegaDeTH* posee una desventaja más bien práctica: no posee manera alguna de controlar la distribución de los valores que producen sus generadores de datos. En particular, todo constructor del tipo de datos para el cual se deriva un generador aleatorio posee la misma probabilidad de ser generado. Esto acarrea una fuerte dependencia entre la distribución de valores generados y la definición particular del tipo de datos a generar, lo cual produce en muchos casos una tendencia hacia generar valores triviales y de escaso interés para el proceso de testing aleatorio. La siguiente figura ilustra este comportamiento para el caso de un generador aleatorio derivado usando *MegaDeTH* para el tipo de datos $Tree_{3L}$ con el que se generaron 100.000 valores utilizando un tamaño de generación igual a 10.

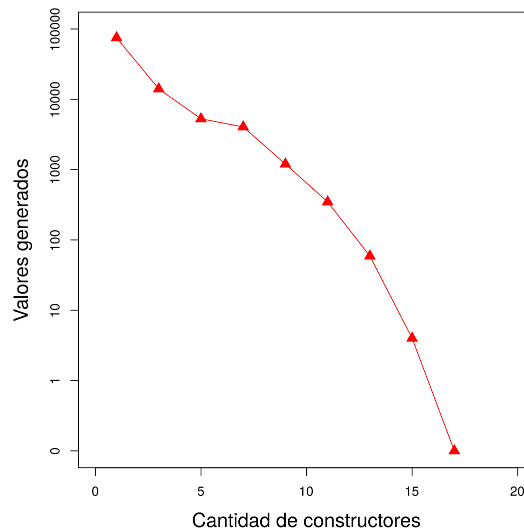


Figura 3.3: Distribución de valores obtenida utilizando el generador obtenido con *MegaDeTH* para el tipo de datos $Tree_{3L}$ y un tamaño de generación igual a 10.

En la figura anterior se puede observar cómo este generador de datos es incapaz de producir

valores aleatorios diversos. En particular, puesto que este tipo de generadores asigna a cada constructor la misma probabilidad de ser generado, en nuestro ejemplo obtendremos una distribución de valores compuesta por hojas en alrededor de un 75 %. Esto además acarrea una tendencia a generar árboles muy poco profundos, ya que la probabilidad de generar n constructores *Node* de manera sucesiva es de $p^n_{Node} = 0.25^n$. Así, generar un árbol de profundidad $n = 10$ posee una probabilidad de 0.000000954. Si consideramos además que estos valores serán luego utilizados para validad propiedades posiblemente complejas sobre sistemas no triviales, generar repetidamente y en tan alta proporción los mismos valores triviales resulta en una performance de testing extremadamente limitada.

3.3. Un enfoque distinto: *Feat*

El último mecanismo de derivación automática de generadores de datos aleatorios en Haskell que presentamos en este trabajo es el que adopta *Feat* [12] (por *Functional Enumeration of Algebraic Types*). A diferencia de *derive* y *MegaDeTH*, donde el proceso de generación aleatoria de datos se formula eligiendo aleatoriamente un constructor de tipos, y generando recursivamente los sub-términos necesarios, *Feat* utiliza un enfoque completamente distinto. Este mecanismo provee una generación uniforme de datos basándose en una *enumeración exhaustiva* de todos los posibles valores del tipo de datos que se desea generar aleatoriamente. Esto es, *Feat* establece una biyección entre un prefijo finito de los números naturales y los valores de un tipo de datos dado.

De esta manera, una enumeración del tipo de datos $(Bool, Bool)$ (el tipo de los pares de booleanos) tiene la siguiente forma:

$$0 \mapsto (True, True) \qquad 1 \mapsto (False, True) \qquad 2 \mapsto (True, False) \qquad 3 \mapsto (False, False)$$

Luego, podemos generar uniformemente valores de este tipo de datos simplemente generando de manera uniforme un número natural entre 0 y 3 y aplicando este mapeo para obtener el valor concreto. Por supuesto, esta enumeración se vuelve más compleja cuando se involucran tipos de datos recursivos y posiblemente infinitos, ya que no es posible generar uniformemente el conjunto completo de los números naturales. Para subsanar esta restricción, *Feat* utiliza el tamaño de generación como parámetro para establecer un prefijo de los números naturales. En particular, el tamaño de generación es interpretado como el número máximo de cons-

tructores que puede tener cualquier valor generado aleatoriamente. Así, todo valor generado usando un tamaño de generación n será construido a partir de (a lo sumo) n constructores. De esta manera, se restringe el conjunto de los posibles valores que pueden ser generados a un conjunto finito, donde es posible nuevamente establecer un mapeo entre éstos y un prefijo de los números naturales. Luego, usando un tamaño de generación n, *Feat* garantiza una generación aleatoria uniforme entre todos los posibles valores del tipo de datos deseado construidos usando hasta n constructores de tipo.

Si bien este enfoque nos permite generar datos de manera uniforme, algo que no se puede conseguir usando *derive* o *MegaDeTH*, las distribuciones de valores—en términos de cantidad de constructores presentes en cada uno—siguen siendo altamente dependientes de cada tipo de datos particular que sea considerado. La siguiente figura ilustra la distribución de valores del generador aleatorio derivado usando *Feat* para el tipo $Tree_{3L}$, con el que se generaron 100.000 valores utilizando un tamaño de generación igual a 400 [**] :

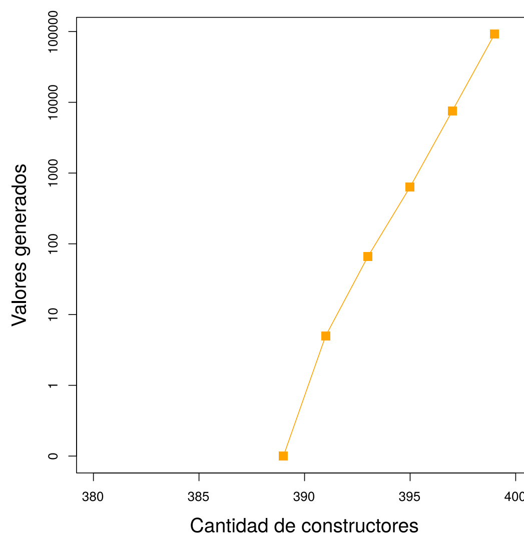


Figura 3.4: Distribución de valores obtenida al utilizar el generador obtenido con *Feat* para el tipo de datos $Tree_{3L}$ y un tamaño de generación igual a 400.

Se puede notar cómo todos los valores generados utilizando este generador aleatorio se en-

[**]Dado que no podemos comparar generadores de datos derivados mediante *MegaDeTH* y *Feat* usando el mismo tamaño de generación debido a la diferente interpretación semántica que éstos le dan al mismo, se escogió usar este tamaño de generación con el fin de comparar resultados con nuestro enfoque de generación. Esta comparación es abordada en el Capítulo 5.

cuentran muy cercanos al tamaño máximo de generación. Este fenómeno es una consecuencia directa del hecho de que el número de posibles valores del tipo $Tree_{3L}$ con n constructores crece exponencialmente a medida que se agranda n. En otras palabras, el espacio de valores del tipo $Tree_{3L}$ de hasta 400 constructores está formado en gran medida por valores de alrededor de 400 constructores, y de muy pocos (en proporción) valores compuestos por un número pequeño de constructores. Dada esta observación, un proceso de generación de valores basado únicamente en la generación aleatoria de un número natural, el cual ignora o pierde toda noción de estructura del tipo en cuestión, está fuertemente sesgado a generar valores muy grandes en presencia de tipos de datos recursivos. En nuestros experimentos, ningún valor del tipo $Tree_{3L}$ construido con menos de 390 constructores fue siquiera generado.

En este trabajo buscamos definir un método de derivación automática de generadores de datos aleatorios basados en la definición del tipo de datos a generar, y que nos permita ajustar la distribución generada de constructores para satisfacer ciertas demandas que el programador pudiese tener. El siguiente capítulo introduce el principal aporte de este trabajo: un mecanismo analítico que nos permite predecir la distribución de constructores obtenida por un tipo particular de generadores de datos, modelándolos a partir de un tipo particular de proceso estocástico.

4

Marco Teórico

En teoría de la probabilidad, los *procesos de ramificación*[19] constituyen un objeto matemático perteneciente al grupo de los procesos estocásticos. Los mismos nacen como un método capaz de modelar el crecimiento y extinción de poblaciones en las que cada individuo perteneciente una generación n produce un número aleatorio de descendientes en la generación $n+1$ de acuerdo a, en el caso más sencillo, una distribución de probabilidad fija y que no varía de individuo a individuo. En este trabajo hacemos uso de un caso particular de los procesos de ramificación, conocidos como *procesos de ramificación de Galton-Watson*, los cuales surgen en 1875 a partir del trabajo[37] de Francis Galton y Henry William Watson. Este trabajo surge como una solución matemática al problema planteado por la aristocracia Victoriana en torno a la posible extinción de los apellidos reales. Desde entonces, este modelo ha sido aplicado exitosamente a diversas áreas de investigación como la biología y la física. En este capítulo se muestra cómo es posible aplicar esta teoría con el fin de modelar el proceso de generación aleatoria de datos mediante *QuickCheck* y generadores de datos derivados automáticamente.

4.1. Procesos de Ramificación de Tipo Simple

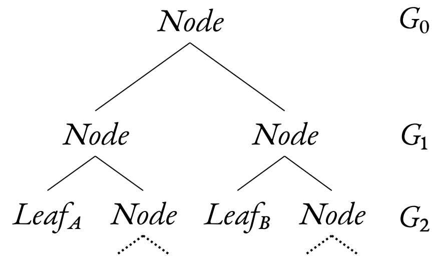
Para comprender cómo se formulan los procesos de ramificación y cuál su relación con la generación aleatoria de datos, es conveniente comenzar analizando el caso más simple: modelar el proceso de generación de los constructores recursivos en aquellos tipos de datos donde sólo existe un constructor de esta naturaleza. Para esto, reutilizamos tipo de datos $Tree_{3L}$ definido anteriormente

$$\texttt{data}\ Tree_{3L} = Leaf_A \mid Leaf_B \mid Leaf_C \mid Node\ Tree_{3L}\ Tree_{3L}$$

junto a su correspondiente generador de datos derivado automáticamente por *MegaDeTH* como se mostró en la Figura 3.2. En este caso, estudiaremos el proceso de generación del constructor *Node*. Guiándonos por el código fuente, podemos observar que el proceso estocástico que gobierna la generación aleatoria de datos satisface las siguientes hipótesis, las cuales coinciden con las hipótesis de los procesos de ramificación de Galton-Watson:

- Con una cierta probabilidad, el proceso comienza generando un constructor *Node*.
- En cualquier momento, la probabilidad de generar un constructor *Node* no depende de los constructores *Node* generados anteriormente.
- La probabilidad de generar un constructor *Node* es independiente de la posición en el árbol que éste ocupará.

Luego, los procesos de ramificación de Galton-Watson de tipo simple nos permiten estimar los tamaños de cada *población* de individuos de la misma especie en distintos puntos de tiempo llamados *generaciones*. Para nuestros propósitos, podemos considerar a las poblaciones como el conjunto de constructores *Node* en cada generación, donde cada generación a su vez se corresponde con cada nivel del árbol generado. La siguiente figura ilustra un posible valor del tipo de datos $Tree_{3L}$ generado aleatoriamente mediante el generador de datos antes citado.



En la figura anterior, el proceso aleatorio comienza generando un constructor *Node* en la generación 0, denotada como G_0, luego se generan dos constructores *Node* como sub-árboles en la generación 1 (G_1), y así sucesivamente. De lo anterior se desprende que la población en la generación 0 se compone de un único individuo; la población en la generación 1 posee dos individuos y así sucesivamente. En general, cada constructor *Node* de una generación (G_i) puede producir hasta dos constructores *Node* en la generación G_{i+1}, a los que llamamos *descendientes*. Las aristas punteadas (⋯⋯) denotan el resto de los constructores generados para este valor, pero que no son esenciales para el punto en cuestión. Este proceso se repite hasta que la población de constructores *Node* en la generación actual se extingue, debido a que o bien no se genera aleatoriamente ningún constructor *Node*, o bien a que se alcanza el límite máximo profundidad (dependiente del tamaño de generación).

La matemática detrás de los procesos de ramificación de Galton-Watson de tipo simple nos permite predecir el número esperado de descendientes en la n-ésima generación. En otras palabras, nos permite predecir el número esperado de constructores *Node* presentes en el nivel n de cada árbol generado aleatoriamente. Formalmente, comenzamos introduciendo la variable aleatoria R para denotar el número de constructores *Node* en la siguiente generación, producidos por un constructor *Node* en la generación actual. Luego, con el fin de aprovechar toda la generalidad disponible en los procesos de ramificación, consideremos que la probabilidad de generar cada constructor ya no está uniformemente distribuida, sino que cada constructor C del tipo de datos $Tree_{3L}$ posee una probabilidad p_C de ser generado aleatoriamente cada vez que generamos un sub-término recursivamente. A su vez, notamos como p_{Leaf} a la probabilidad de generar una hoja de cualquier clase. Es decir, $p_{Leaf} = p_{Leaf_A} + p_{Leaf_B} + p_{Leaf_C}$. De esta manera, y asumiendo que se tiene un constructor *Node* como padre, las probabilidades de generar R constructores *Node* en la siguiente generación mediante llamadas recursivas a *arbitrary* resultan como se muestra a continuación:

$$P(R=0) = p_{Leaf} \cdot p_{Leaf}$$
$$P(R=1) = p_{Node} \cdot p_{Leaf} + p_{Leaf} \cdot p_{Node} = 2 \cdot p_{Node} \cdot p_{Leaf}$$
$$P(R=2) = p_{Node} \cdot p_{Node}$$

Una vez determinada la distribución de la variable aleatoria R, procedemos a introducir la familia de variables aleatorias G_n, que denotan la población de constructores *Node* en la n-ésima generación. Además, escribimos ξ_i^n para denotar a la variable aleatoria que captura el número de constructores *Node* en la generación n que descienden del i-ésimo constructor *Node* en la generación $(n-1)$. Así, es fácil ver que $G_n = \xi_1^n + \xi_2^n + \cdots + \xi_{G_{n-1}}^n$. Luego, para deducir $E[G_n]$, es decir, la cantidad esperada de constructores *Node* en la generación n, procedemos a aplicar la *ley de esperanzas iteradas* $E[X] = E[E[X|Y]]$ con $X = G_n$ y $Y = G_{n-1}$, obteniendo:

$$E[G_n] = E[E[G_n|G_{n-1}]] \tag{4.1}$$

Si expandimos G_n, podemos deducir que:

$$\begin{aligned} E[G_n|G_{n-1}] &= E[\xi_1^n + \xi_2^n + \cdots + \xi_{G_{n-1}}^n|G_{n-1}] \\ &= E[\xi_1^n|G_{n-1}] + E[\xi_2^n|G_{n-1}] + \cdots + E[\xi_{G_{n-1}}^n|G_{n-1}] \end{aligned}$$

Dado que ξ_1^n, ξ_2^n, $\cdots$, y $\xi_{G_{n-1}}^n$ están *todas gobernadas por la distribución capturada por la variable aleatoria R* (recordar las hipótesis al principio de este capítulo), tenemos que:

$$E[G_n|G_{n-1}] = E[R|G_{n-1}] + E[R|G_{n-1}] + \cdots + E[R|G_{n-1}]$$

Y dado que R es *independiente de la generación en que un constructor Node genera otro constructor Node*, entonces tenemos que:

$$E[G_n|G_{n-1}] = \underbrace{E[R] + E[R] + \cdots + E[R]}_{G_{n-1} \text{ veces}} = E[R] \cdot G_{n-1} \tag{4.2}$$

De aquí en más, denotaremos como m a la media de R, es decir, *la media de reproducción*. Introduciendo $m = E[R]$ y reescribiendo en la ecuación anterior se obtiene:

$$E[G_n] \underset{\text{por (4.1)}}{=} E[E[G_n|G_{n-1}]] \underset{\text{por (4.2) y def. de } m}{=} E[m \cdot G_{n-1}] \underset{m \text{ es constante}}{=} E[G_{n-1}] \cdot m$$

Y por inducción en n obtenemos:

$$E[G_n] = E[G_0] \cdot m^n \tag{4.3}$$

Como indica la ecuación anterior, la cantidad esperada de constructores *Node* en la n-ésima generación es afectada por la media de reproducción.

Con los resultados anteriores, estamos en condiciones de predecir la cantidad esperada total de individuos *hasta la n-ésima generación*. Para esto, introducimos la variable aleatoria P_n que denota la población de constructores *Node* hasta la n-ésima generación. Es fácil notar que $P_n = \sum_{i=0}^{n} G_i$, y por ende:

$$E[P_n] = \sum_{i=0}^{n} E[G_i] \underset{\text{por (4.3)}}{=} \sum_{i=0}^{n} E[G_0] \cdot m^i \underset{\text{por serie geom.}}{=} E[G_0] \cdot \left(\frac{1 - m^{n+1}}{1 - m} \right) \tag{4.4}$$

De esta manera, deducimos la fórmula provista por la teoría de los procesos de ramificación de Galton-Watson aplicada a nuestro propósito particular. Luego, la media de reproducción para el constructor *Node* viene dada por:

$$E[R] = \sum_{k=0}^{2} k \cdot P(R = k) = 2 \cdot p_{Node} \tag{4.5}$$

Y por (4.4) y (4.5), la cantidad esperada de constructores *Node* hasta la generación n viene dada por la fórmula:

$$E[P_n] = E[G_0] \cdot \left(\frac{1 - (2 \cdot p_{\text{Node}})^{n+1}}{1 - 2 \cdot p_{\text{Node}}} \right) = p_{\text{Node}} \cdot \left(\frac{1 - (2 \cdot p_{\text{Node}})^{n+1}}{1 - 2 \cdot p_{\text{Node}}} \right)$$

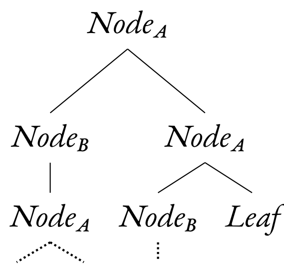
Por ejemplo, si asignamos $p_{Leaf_A} = p_{Leaf_B} = p_{Leaf_C} = 0.1$ y $p_{Node} = 0.7$, podemos prede-

cir que el generador aleatorio derivado por *MegaDeTH* generará [*] árboles con una cantidad promedio de 69.1175 constructores *Node* hasta el nivel 10.

Desafortunadamente, no es posible aplicar este razonamiento para predecir la distribución de constructores para aquellos tipos de datos con más de un constructor no-terminal. Por ejemplo, consideremos el siguiente tipo de datos:

$$\texttt{data}\ Tree_{2N} = Leaf \mid Node_A\ Tree_{2N}\ Tree_{2N} \mid Node_B\ Tree_{2N}$$

En este caso hace falta considerar por separado los casos en que un constructor $Node_A$ genera no sólo constructores $Node_A$ sino también constructores $Node_B$ (y viceversa considerando $Node_B$). La siguiente figura ilustra un posible valor generado aleatoriamente que sigue este comportamiento.



Para lograr modelar este proceso de generación aleatoria hace falta un formalismo más potente. La siguiente sección aborda la manera de predecir la generación aleatoria de datos pertenecientes a tipos de datos con múltiples constructores no-terminales mediante el uso de una extensión a los procesos de ramificación de Galton-Watson conocida como *procesos de ramificación multi-tipo*.

[*] Siempre que se utilice un tamaño de generación suficientemente grande como para no abortar la generación antes de llegar a dicha profundidad.

4.2. Procesos de Ramificación Multi-Tipo

Hasta ahora sólo hemos sido capaces de predecir la distribución de constructores no-terminales para aquellos tipos de datos con un único constructor de esta naturaleza. En el caso general, distintos constructores no-terminales podrían, en principio, producir diferentes tipos de descendientes con distintas probabilidades. En esta sección se aborda la base de nuestra contribución principal: *el uso de los procesos de ramificación de Galton-Watson multi-tipo para predecir la distribución de constructores generados aleatoriamente*. Ilustraremos el desarrollo teórico de los mismos utilizando como ejemplo motivador el tipo de datos $Tree_{2N}$ introducido al final de la sección anterior como testigo de los límites teóricos de los procesos de ramificación de tipo simple. En primer lugar, introducimos a continuación una versión simplificada, aunque semánticamente equivalente, del generador de datos aleatorios para el tipo de datos $Tree_{2N}$ derivado automáticamente por nuestra herramienta, a la que llamamos *DRaGen* (por *Derivation of Random Generators*).

```
$(dragenArbitrary ''Tree2N 10 uniform)
 ⟹
instance Arbitrary Tree2N where
  arbitrary = sized gen
    where
    gen 0  = genTerm
    gen n  = genAny n
    genTerm = return Leaf
    genAny n = do
      t1 ← resize (n − 1) arbitrary
      t2 ← resize (n − 1) arbitrary
      genWithProb
        [ (p_Leaf,  Leaf)
        , (p_NodeA, NodeA t1 t2)
        , (p_NodeB, NodeB t1)]
```

Figura 4.1: Generador aleatorio para el tipo de datos $Tree_{2N}$ derivado por nuestra herramienta.

Por ahora, no hace falta prestar especial atención a los parámetros usados al derivar este generador de datos (el número 10 y *uniform*). Los mismos serán abordados en profundidad en el Capítulo 5. Como es de esperarse, nuestro generador derivado automáticamente utiliza la función *sized* para controlar la profundidad máxima de generación. Sin embargo, y a diferencia de *MegaDeTH* donde el tamaño de generación decrece de manera no constante, en nuestro generador siempre se decrementa el tamaño de generación en una unidad cuando se efectúan llamadas recursivas. Esta es una pieza clave de nuestro proceso predictivo, ya que nos permite establecer una relación fija entre el tamaño de generación y la profundidad máxima de los valores generados, la cual utilizaremos en la siguiente sección para calcular la cantidad de constructores terminales generados. De esta manera, todo valor generado aleatoriamente utilizando un tamaño de generación *n* tendrá una profundidad de *hasta n* niveles. Por otro lado, nuestros generadores dependen de las probabilidades (posiblemente) distintas que cada constructor tiene de ser generado—ver variables p_{Leaf}, p_{Node_A} y p_{Node_B}. Las mismas son usadas por la función $genWithProb :: [(Double, a)] \rightarrow Gen\ a$ que elige con una probabilidad explícita un valor de tipo *a* de una lista dada de valores. Esta función se puede expresar fácilmente utilizando las primitivas[**] que ofrece *QuickCheck* y es por eso que obviamos enfocarnos en su implementación. Finalmente, podría pensarse que nuestro generador se encarga de generar recursivamente dos sub-árboles antes de siquiera decidir qué constructor será generado. Este comportamiento no degrada en absoluto la performance del proceso de generación, ya que *QuickCheck* (y en particular el tipo monádico *Gen*) explota la capacidad de realizar evaluación lazy en Haskell. De esta manera, estos sub-árboles sólo son generados cuando es estrictamente necesario, es decir, cuando se generan nodos. Estas consideraciones sobre nuestros generadores derivados automáticamente nos aseguran que los mismos satisfacen las hipótesis de los procesos de ramificación antes vistas.

Procederemos ahora a modelar la distribución de constructores inducida por nuestros generadores de datos aleatorios mediante procesos de ramificación multi-tipo, los cuales pueden verse como una generalización de los procesos de ramificación de tipo simple antes vistos. Esta generalización nos permite modelar el hecho de que varios tipos de individuos (constructores) pueden procrear (generar) varios tipos de descendientes (constructores). Además, este enfoque es capaz de considerar no sólo una clase de constructores a la vez, sino todos en simultáneo.

[**] Utilizando por ejemplo la función de *QuickCheck frequency*.

A continuación presentamos las bases matemáticas de este modelo, las cuales siguen una línea de razonamiento similar a la presentada en la sección anterior.

En el proceso de generación, se asume que *la probabilidad de que un constructor de cierta variedad sea generado puede depender de la variedad del constructor padre que lo genera*. Observar que este el el caso para la mayoría de los tipos de datos que se encuentran comúnmente. Por ejemplo, generar un constructor terminal (como *Leaf*) afecta la probabilidad de generar un constructor no-terminal (llevándola a cero).

Para caracterizar la n-ésima población usaremos un vector de variables aleatorias $G_n = (G_n^1, G_n^2, \cdots, G_n^d)$, donde d es el número de posibles descendientes diferentes, es decir, el número de constructores involucrados en el proceso de generación aleatoria. Luego, la variable aleatoria G_n^i se encarga de capturar el número de ocurrencias del i-ésimo constructor en la n-ésima generación. De esta manera, G_n esencialmente "agrupa" el nivel de población n según el tipo de constructor, y estimando la forma esperada del vector G_n es posible obtener el número esperado de cada constructor distinto en la n-ésima generación. Específicamente, tenemos que:

$$E[G_n] = (E[G_n^1], E[G_n^2], \cdots, E[G_n^d])$$

Para deducir $E[G_n]$, nos enfocamos en deducir cada componente del vector. Como se explicó anteriormente, la reproducción está determinada por la variedad tanto del constructor hijo como del constructor padre. Para esto, introducimos la variable aleatoria R_{ij} que denota al i-ésimo constructor como padre, generando un j-ésimo constructor como hijo. Como hicimos antes, aplicamos la ecuación $E[X] = E[E[X|Y]]$ con $X = G_n^j$ y $Y = G_{n-1}$ obteniendo:

$$E[G_n^j] = E[E[G_n^j | G_{n-1}]]$$

Para deducir $E[G_n^j | G_{n-1}]$, notamos que el número esperado de constructores j-ésimos en la generación n es el número esperado de constructores j-ésimos producidos por los diferentes constructores padres en la generación $(n-1)$. Formalmente:

$$E[G_n^j | G_{n-1}] = \sum_{i=1}^{d} G_{n-1}^i \cdot E[R_{ij}] \tag{4.6}$$

Luego, por simplicidad escribimos $E[R_{ij}]$ como m_{ij}. En la sección anterior, m representaba la media de reproducción de un constructor único. Ahora m_{ij} representa la media de reproducción indexada por los constructores padre e hijo. Observar que ahora tenemos una colección de valores medios de reproducción, es decir, $\{m_{ij}\}_{i,j=1}^{d}$, a los que llamaremos la *matriz de reproducción de constructores* M_C. Esta matriz es un parámetro de los procesos de ramificación de Galton-Watson multi-tipo. Luego, estamos en posición de deducir $E[G_n^j]$.

$$
\begin{aligned}
E[G_n^j] \underset{\text{por prob.}}{=} E[E[G_n^j|G_{n-1}] \underset{\text{por (4.6)}}{=} E\left[\sum_{i=1}^{d} G_{n-1}^i \cdot m_{ij}\right] \\
\underset{\text{por prob.}}{=} \sum_{i=1}^{d} E[G_{n-1}^i \cdot m_{ij}] = \sum_{i=1}^{d} E[G_{n-1}^i] \cdot m_{ij}
\end{aligned}
$$

Y usando esta última ecuación podemos reescribir $E[G_n]$ como sigue:

$$
E[G_n] = \left(\sum_{i=1}^{d} E[G_{n-1}^1] \cdot m_{i1}, \sum_{i=1}^{d} E[G_{n-1}^2] \cdot m_{i2}, \cdots, \sum_{i=1}^{d} E[G_{n-1}^d] \cdot m_{id}\right)
$$

Por álgebra lineal, este vector puede reescribirse como la siguiente multiplicación matricial:

$$
E[G_n]^T = E[G_{n-1}]^T \cdot M_C
$$

Y aplicando nuevamente inducción en n obtenemos:

$$
E[G_n]^T = E[G_0]^T \cdot M_C^n \tag{4.7}
$$

Esta ecuación es una generalización de (4.3) considerando una variedad de constructores distintos. Como hicimos antes, introducimos la variable aleatoria $P_n = \sum_{i=0}^{n} G_i$ para denotar la población de constructores hasta la n-ésima generación. Ahora resulta posible obtener la población esperada de los distintos constructores de manera agrupada:

$$
E[P_n]^T \underset{\text{por def.}}{=} E\left[\sum_{i=0}^{n} G_i\right]^T \underset{\text{por prob.}}{=} \sum_{i=0}^{n} E[G_i]^T \underset{\text{por (4.7)}}{=} \sum_{i=0}^{n} E[G_0]^T \cdot M_C^n \tag{4.8}
$$

Y es posible reescribir esta última sumatoria utilizando la serie geométrica generalizada para matrices como sigue:

$$E[P_n]^T \underset{\text{por serie geom.}}{=} E[G_0]^T \cdot \left(\frac{I - M_C^{n+1}}{I - M_C} \right) \tag{4.9}$$

Donde el símbolo I denota la matriz identidad del tamaño correspondiente. Notar que la ecuación anterior sólo es aplicable siempre que $(I - M_C)$ sea no-singular (lo cual es el caso usual). Cuando $(I - M_C)$ es singular, resolvemos utilizar la ecuación (4.8) en su lugar. Sin perder generalidad, y por cuestiones de simplicidad, consideraremos a las ecuaciones (4.8) y (4.9) como intercambiables. Esta es la fórmula general provista por los procesos de ramificación de Galton-Watson multi-tipo.

La matriz de reproducción de constructores puede construirse siguiendo el mismo procedimiento que realizamos en la sección anterior para calcular m, pero considerando ahora la variedad de los constructores padre e hijo en cada caso:

$$M_C = \left[m_{ij} \right]_{i,j \,\in\, \{Leaf, Node_A, Node_B\}} = \begin{array}{c} \\ Leaf \\ Node_A \\ Node_B \end{array} \begin{array}{c} \begin{array}{ccc} Leaf & Node_A & Node_B \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 2 \cdot p_{Leaf} & 2 \cdot p_{Node_A} & 2 \cdot p_{Node_B} \\ p_{Leaf} & p_{Node_A} & p_{Node_B} \end{bmatrix} \end{array} \tag{4.10}$$

Donde m_{ij} es el número esperado de constructores j-ésimos generados por el i-ésimo constructor. Observar en particular que la primer fila de M_C esta compuesta por ceros. Esto se debe a que *Leaf* es un constructor terminal, y por ende no puede generar sucesivos constructores de ninguna clase. Para calcular el resto de los elementos de M_C, hace falta realizar algunos cálculos adicionales de valores esperados. Por ejemplo, el número esperado de constructores *Leaf* generados por un constructor $Node_A$ se deduce como sigue (el resto de los elementos se deducen de manera análoga).

$$
\begin{aligned}
m_{Node_A,\, Leaf} &= \underbrace{1 \cdot p_{Leaf} \cdot p_{Node_A} + 1 \cdot p_{Leaf} \cdot p_{Node_B}}_{\text{Una hoja como sub-árbol izquierdo}} \\
&+ \underbrace{1 \cdot p_{Node_A} \cdot p_{Leaf} + 1 \cdot p_{Node_B} \cdot p_{Leaf}}_{\text{Una hoja como sub-árbol derecho}} \\
&+ \underbrace{2 \cdot p_{Leaf} \cdot p_{Leaf}}_{\text{Dos hojas como sub-árboles izquierdo y derecho}} \\
&= 2 \cdot p_{Leaf}
\end{aligned}
$$

Un lector atento puede haber notado que existe una relación implícita entre la matriz M_C y la definición del tipo de datos $Tree_{2N}$. En la Sección 4.5 probamos utilizando esta relación que cada elemento m_{ij} puede ser automáticamente calculado utilizando la información presente en la definición del tipo de datos en cuestión.

Una vez obtenida la matriz de reproducción de constructores, definimos el vector de valores esperados iniciales para la primer generación como $E[G_0] = (p_{Leaf}, p_{Node_A}, p_{Node_B})$. Y aplicando la ecuación (4.9) con $E[G_0]$ y M_C antes definida, podemos predecir el número promedio de constructores *no-terminales* generados utilizando un tamaño de generación n mediante nuestro generador de datos derivado automáticamente como se muestra a continuación.

$$
E[\text{Node}_\text{A}] = \left(E[P_{n-1}]^T\right).\text{Node}_\text{A} = \left(E[G_0]^T \cdot \left(\frac{I - M_C^n}{I - M_C}\right)\right).\text{Node}_\text{A}
$$
$$
E[\text{Node}_\text{B}] = \left(E[P_{n-1}]^T\right).\text{Node}_\text{B} = \left(E[G_0]^T \cdot \left(\frac{I - M_C^n}{I - M_C}\right)\right).\text{Node}_\text{B}
$$

Donde la función $(_).C$ simplemente proyecta el valor correspondiente al constructor C del vector de población. Es importante remarcar en las ecuaciones anteriores que la suma de generaciones sólo incluye las primeras $(n-1)$ generaciones (ver $E[P_{n-1}]$). Esto se debe a que, como vimos antes, nuestro generador sólo puede elegir generar constructores no-terminales en tanto no se haya alcanzado el límite de profundidad de generación, el cual se corresponde con n. Así, la n-ésima generación está compuesta únicamente por constructores terminales, y en nuestro caso particular está compuesta únicamente por constructores *Leaf*. Como ejemplo,

si asignamos las probabilidades $p_{Leaf} \mapsto 0.2$, $p_{Node_A} \mapsto 0.5$ y $p_{Node_B} \mapsto 0.3$, luego podemos predecir que nuestro generador aleatorio derivado automáticamente para el tipo de $Tree_{2N}$ usando un tamaño de generación igual a 10 generará en promedio 21.322 constructores $Node_A$ y 12.813 constructores $Node_B$. Este resultado puede verificarse fácilmente generando un conjunto grande de valores usando este generador de datos, utilizando el tamaño de generación antes citado, y luego promediando el número de constructores $Node_A$ y $Node_B$ presentes en todos los valores generados aleatoriamente.

En esta sección mostramos como predecir analíticamente el número esperado de constructores no-terminales generados aleatoriamente mediante nuestros generadores de datos derivados automáticamente. Predecir el número esperado de constructores terminales, sin embargo, requiere un tratamiento especial abordado por la siguiente sección.

4.3. Predicción de Constructores Terminales

Hasta aquí hemos sido capaces de predecir el comportamiento de nuestros generadores de datos derivados automáticamente para aquellos constructores no-terminales. Para predecir la cantidad de constructores terminales generados por nuestros generadores aleatorios debemos considerar que los mismos generan constructores terminales en dos situaciones distintas: i) en las primeras $(n-1)$ generaciones, siguiendo el modelo descripto mediante procesos de ramificación visto anteriormente, y ii) en la última generación n, donde se alcanzó el límite de profundidad dado por el tamaño de generación, y sólo se generan constructores terminales para "llenar" los argumentos recursivos de aquellos constructores no-terminales producidos en la generación anterior. Si analizamos el código fuente del generador aleatorio derivado automáticamente para el tipo de datos $Tree_{2N}$ visto en la Figura 4.1, podemos observar que ambas situaciones se corresponden con las funciones auxiliares $genTerm$ y $genAny$, respectivamente. La siguiente figura ilustra gráficamente el proceso completo de generación aleatoria de datos, diferenciando ambas situaciones antes descriptas.

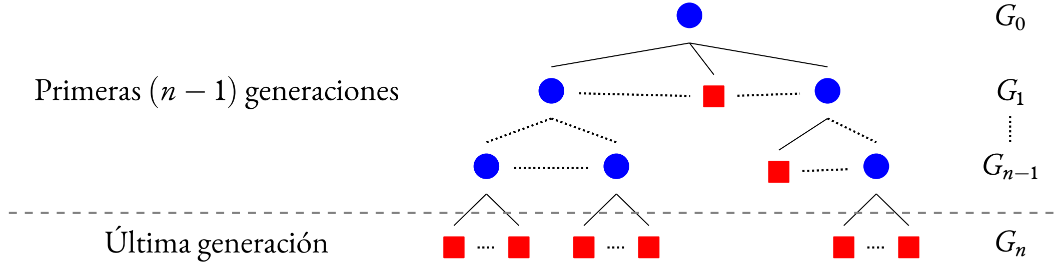


Figura 4.2: Proceso de generación de constructores terminales (■) y no-terminales (●).

Siguiendo la figura anterior, es fácil deducir que para predecir el número de constructores terminales generados durante el proceso completo de generación simplemente hace falta sumar la cantidad de constructores terminales generados por ambas etapas de generación. De esta manera, es posible por ejemplo predecir la cantidad de constructores *Leaf* generados mediante nuestro generador aleatorio para el tipo de datos $Tree_{2N}$ usando un tamaño de generación n. Para esto, combinamos la cantidades esperadas de constructores *Leaf* producidos hasta la generación $(n-1)$ mediante el proceso de ramificación antes descripto, junto con los constructores *Leaf* necesarios para llenar los argumentos recursivos de los constructores $Node_A$ y $Node_B$ presentes en la generación $(n-1)$. Formalmente:

$$E[Leaf] = \underbrace{\left(E[P_{n-1}]^T\right).Leaf}_{\text{por } (4.9)} + \underbrace{2 \cdot \left(E[G_{n-1}]^T\right).\text{Node}_\text{A}}_{\text{Caso } (Node_A\ Leaf\ Leaf)} + \underbrace{\left(E[G_{n-1}]^T\right).\text{Node}_\text{B}}_{\text{Caso } (Node_B\ Leaf)} \qquad (4.11)$$

Si bien ahora somos capaces de predecir la cantidad esperada de constructores del tipo de datos $Tree_{2N}$ generados aleatoriamente, sin importar si son o no constructores terminales, éste enfoque sólo funciona para tipos de datos con un único constructor terminal. En presencia de tipos de datos con múltiples constructores terminales, hace falta además considerar las probabilidades de elegir cada uno de los constructores terminales disponibles al momento de llenar los argumentos recursivos de los constructores no-terminales presentes en la generación anterior. De esta manera, podemos pensar que la generación de constructores terminales está dictada por dos procesos aleatorios diferentes que dependen del nivel de profundidad de cada constructor generado. Consideremos el siguiente tipo de datos con múltiples constructores terminales y no terminales:

```
data Tree_2L2N = Leaf_A | Leaf_B | Node_A Tree_2L2N Tree_2L2N | Node_B Tree_2L2N
```

Junto a su correspondiente generador aleatorio derivado usando nuestra herramienta:

```
$(dragenArbitrary ''Tree_2L2N)
⟹
instance Arbitrary Tree_2L2N where
  arbitrary = sized gen
    where
      gen 0 = genTerm
      gen n = genAny n
      genTerm = chooseWith
        [ (p*_Leaf_A, Leaf_A)
        , (p*_Leaf_B, Leaf_B) ]
      genAny n = do
        t1 ← resize (n − 1) arbitrary
        t2 ← resize (n − 1) arbitrary
        chooseWith
          [ (p_Leaf_A, Leaf_A)
          , (p_Leaf_B, Leaf_B)
          , (p_Node_A, Node_A t1 t2)
          , (p_Node_B, Node_B t1) ]
```

Lo primero que podemos notar en este generador aleatorio es la existencia de dos juegos de probabilidades para los constructores terminales del tipo de datos $Tree_{2L2N}$, uno por cada proceso aleatorio antes nombrado. En particular, las probabilidades distinguidas usando $(*)$ son utilizadas para elegir aleatoriamente los constructores terminales en el último nivel del proceso de generación. Las mismas preservan las mismas proporciones que sus versiones no-

distinguidas, y se derivan a partir de estas últimas mediante normalización, con el fin de formar una distribución de probabilidad:

$$p^*_{Leaf_A} = \frac{p_{Leaf_A}}{p_{Leaf_A} + p_{Leaf_B}} \qquad p^*_{Leaf_B} = \frac{p_{Leaf_B}}{p_{Leaf_A} + p_{Leaf_B}}$$

De esta manera, nuestro motor predictivo puede utilizar las mismas probabilidades de generación de constructores terminales para ambos procesos aleatorios, reduciendo así el número de parámetros de entrada que deben ser manipulados por el mismo.

Luego, para computar el número esperado de constructores terminales de cada clase generados aleatoriamente en el último nivel, simplemente distribuimos el número esperado de constructores terminales en la última generación mediante sus probabilidades distinguidas. Más precisamente, tenemos que:

$$\begin{aligned} E[Leaf_A] = & \underbrace{\left(E[P_{n-1}]^T\right).\mathrm{Leaf_A}}_{\text{proceso de ramificación multi-tipo}} \\ & + \underbrace{2 \cdot p^*_{Leaf_A} \cdot \left(E[G_{n-1}]^T\right).\mathrm{Node_A}}_{\text{número esperado de } Leaf_A\text{s para llenar constructores } Node_A} \\ & + \underbrace{p^*_{Leaf_A} \cdot \left(E[G_{n-1}]^T\right).\mathrm{Node_A}}_{\text{número esperado de } Leaf_A\text{s para llenar constructores } Node_B} \end{aligned}$$

El caso de $E[Leaf_B]$ se calcula análogamente. Notar que la ecuación (4.11) es un caso particular de este último, dónde sólo existe un constructor terminal, y por ende $p^*_{Leaf} = 1$.

Hasta aquí somos capaces de predecir las distribuciones de constructores en generadores derivados automáticamente mediante nuestra herramienta, para aquellos tipos de datos recursivos y con un número arbitrario de constructores terminales y no-terminales. La siguiente sección presenta una manera de extender nuestro mecanismo predictivo a pares de tipos de datos definidos mediante recursión mutua.

En esta sección se muestra cómo nuestro modelo de predicción basado en procesos de ramificación de Galton-Watson multi-tipo se extiende naturalmente a tipos de datos mutuamente recursivos. Como hicimos antes, ilustraremos este concepto haciendo uso de un ejemplo motivador. Consideremos el siguiente par de tipos mututamente recursivos:

```
data T1 = A | B T1 T2
data T2 = C | D T1
```

Si consideramos a T_1 como el tipo inicial en la generación de datos, es decir, generaremos valores de tipo T_1, nuestra herramienta se encargará de sintetizar tanto generadores aleatorios para T_1 como para T_2 como se muestra a continuación:

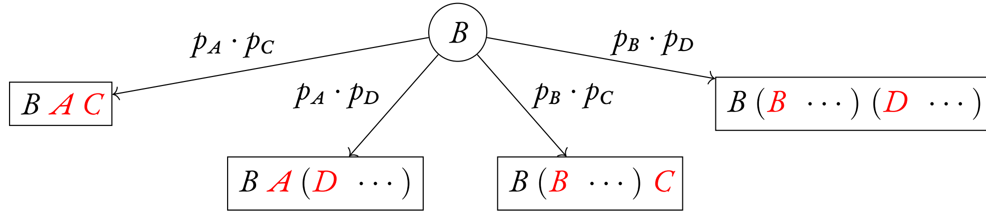
```
$(dragenArbitrary ''T1 10 uniform)
==>
instance Arbitrary T1 where
  arbitrary = sized gen
    where
      gen 0 = genTerm
      gen n = genAny n
      genTerm = return A
      genAny n = do
        t1 <- resize (n - 1) arbitrary
        t2 <- resize (n - 1) arbitrary
        chooseWith
          [(pA, A), (pB, B t1 t2)]

instance Arbitrary T2 where
  arbitrary = sized gen
    where
      gen 0 = genTerm
      gen n = genAny n
      genTerm = return C
      genAny n = do
        t1 <- resize (n - 1) arbitrary
        chooseWith
          [(pC, C), (pD, D t1)]
```

Es interesante notar cómo ambos generadores mostrados arriba hacen llamadas mutuamente recursivas a *arbitrary* para generar los valores aleatorios de sus argumentos mutuamente

recursivos. Allí se puede observar claramente cómo la inferencia de tipos nos permite usar la función sobrecargada *arbitrary* para generar valores de distintos tipos de datos dentro una misma función sin preocuparnos por desambiguar qué instancia concreta debe usarse.

Para predecir la distribución de constructores inducida por este par de generadores mutuamente recursivos, la observación clave reside en que podemos simplemente ignorar que A, B, C y D son *constructores pertenecientes a distintos tipos de datos* y considerar cada uno de ellos como *un tipo de descendiente por si mismo*. Por ejemplo, la siguiente figura ilustra los posibles descendientes inmediatos generados por el constructor no-terminal B perteneciente al tipo T_1, con sus correspondientes probabilidades anotadas en las aristas.



Denotamos mediante $(\cdots)$ aquellos argumentos recursivos que serán producidos en en las sucesivas generaciones. Siguiendo la figura anterior, podemos calcular por ejemplo el número esperado de constructores D generados por un constructor B como se muestra a continuación:

$$m_{BD} = 1 \cdot p_A \cdot p_D + 1 \cdot p_B \cdot p_D = p_D \cdot (p_A + p_B) = p_D$$

Repitiendo este cálculo para el resto de los constructores involucrados, podemos obtener la matriz de reproducción de constructores M_C para A, B, C y D como sigue:

$$M_C = \Big[m_{ij}\Big]_{i,\, j \,\in\, \{A,B,C,D\}} = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \left[\begin{array}{cc|cc} 0 & 0 & 0 & 0 \\ p_A & p_B & p_C & p_D \\ \hline 0 & 0 & 0 & 0 \\ p_A & p_B & 0 & 0 \end{array}\right] \end{array} \tag{4.12}$$

En las secciones anteriores definimos el vector de probabilidades iniciales $E[G_0]$ simplemente

distribuyendo las probabilidades de los constructores del tipo de datos en cuestión. No obstante, en presencia de tipos de datos mutuamente recursivos debemos además considerar que la generación de datos siempre comienza por un tipo de datos fijo—en nuestro ejemplo consideramos a T_1 como el tipo de datos inicial. De esta manera, para definir $E[G_0]$ sólo debemos distribuir las probabilidades del tipo de datos inicial. Para nuestro ejemplo particular resulta $E[G_0] = (p_A, p_B, 0, 0)$.

Una vez deducidos M_C y $E[G_0]$, podemos aplicar las ecuaciones introducidas en las secciones anteriores para predecir el número esperado de constructores A, B, C y D usando nuestros generadores aleatorios.

Si bien este enfoque funciona, para deducir M_C ignoramos completamente la presencia de varios tipos de datos distintos, considerando a los constructores involucrados como todos de un mismo tipo de datos—en nuestro ejemplo ignoramos la existencia T_1 y T_2. Este enfoque puede acarrear un alto costo computacional si deseamos manipular grandes conjuntos de tipos de datos mutuamente recursivos con un gran número de constructores. En particular, operaciones como la multiplicación de matrices de tamaño $n \times n$ poseen una complejidad de orden cúbico en n. En la siguiente sección se introduce una optimización a este modelo, la cual nos permite no sólo reducir el tamaño de la matriz de reproducción de constructores, sino que también nos permite derivarla automáticamente a partir de las definiciones de los tipos de datos en cuestión.

4.5. Matriz de Reproducción de Tipos

En esta sección propondremos una optimización al mecanismo de predicción detallado en las secciones anteriores. En principio, nos detenemos a analizar detenidamente las matrices de medias de reproducción que calculamos para el tipo de datos $Tree_{2N}$ en (4.10) y para los tipos de datos mutuamente recursivos T_1 y T_2 en (4.12). Un observador atento puede notar el siguiente patrón: pareciera que para determinar el número esperado de constructores C_j que descienden de un constructor no-terminal C_i se requiere *contar las ocurrencias del tipo de datos al que C_i pertenece en la definición del constructor C_j*. Por ejemplo, en (4.10) se observa que $m_{Node_A,Leaf}$ es $2 \cdot p_{Leaf}$, donde 2 es el número de ocurrencias del tipo $Tree_{2N}$ (al que pertenece el constructor *Leaf*) en la definición del constructor $Node_A$. De manera similar, en

(4.12) se observa que m_{BD} es $1 \cdot p_D$, donde 1 es el número de ocurrencias del tipo T_2 (al que pertenece el constructor D) en la definición del constructor B. Esta observación nos sugiere que, en lugar de considerar constructores individualmente, ¡podríamos en cambio manipular directamente las definiciones de sus tipos de datos! De esta manera, podemos pensar en un proceso de ramificación que genera *placeholders de tipos*, donde cada placeholder de un tipo de datos dado sólo puede ser "poblado" por constructores que pertenezcan a dicho tipo de datos. Siguiendo este enfoque, la siguiente figura ilustra los posibles placeholders de tipo que descienden inmediatamente de los tipos de datos $Tree_{2N}$, T_1y T_2:

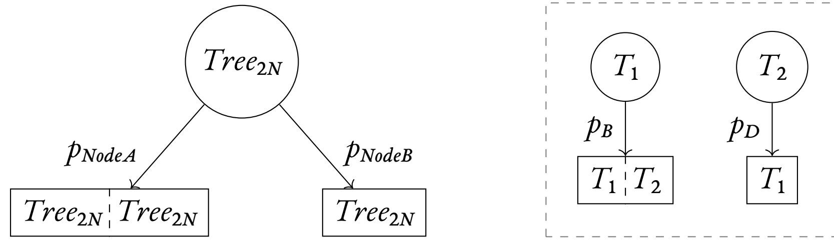


Figura 4.3: Placeholders de tipos como descendientes.

A partir de la figura podemos observar que un placeholder de tipo $Tree_{2N}$ puede generar dos placeholder de tipo $Tree_{2N}$ cuando se genera un constructor $Node_A$, junto con un placeholder de tipo $Tree_{2N}$ cuando se genera un constructor $Node_B$, y cero placeholders de tipo cuando se genera un constructor *Leaf*—este caso no se muestra en la figura ya que es vacío. De manera similar, un placeholder de tipo T_1 puede generar un placeholder de tipo T_1 y un placeholder de tipo T_2 cuando se genera un constructor B, mientras que un placeholder de tipo T_2 puede generar un placeholder de tipo T_1 cuando se genera un constructor D. Con estas consideraciones, y siguiendo el mecanismo usado anteriormente, podemos construir la *matriz de reproducción de tipos* M_T correspondiente al tipo $Tree_{2N}$ como sigue:

$$M_T = \Big[m_{ij}\Big]_{i,\, j \,\in\, \{Tree_{2N}\}} = \begin{array}{c} \\ {\scriptstyle Tree_{2N}} \end{array} \begin{array}{c} {\scriptstyle Tree_{2N}} \\ \left[\; 2 \cdot p_{NodeA} + p_{NodeB} \; \right] \end{array} \tag{4.13}$$

Este ejemplo muestra cómo las matrices de medias de reproducción de tipos de datos nos permiten simplificar nuestros modelos basados en procesos de ramificación de Galton-Watson multi-tipo a unos de tipo simple, siempre que estemos en presencia de tipos de datos recursivos simples (no mutuamente recursivos). De la misma manera, a partir de la Figura 4.3 po-

demos construir la matriz de reproducción de tipos para el caso de los tipos de datos mutuamente recursivos T_1 y T_2 como sigue:

$$M_T = \left[m_{ij} \right]_{i,\, j \,\in\, \{T1, T2\}} = \begin{array}{c} \\ T1 \\ T2 \end{array} \begin{array}{c} \begin{array}{cc} T1 & T2 \end{array} \\ \left[\begin{array}{cc} p_B & p_B \\ p_D & 0 \end{array} \right] \end{array} \tag{4.14}$$

El siguiente teorema, cuya prueba se muestra en el Apéndice A, muestra cómo podemos utilizar las matrices de reproducción de tipos para predecir de manera correcta el número esperado de constructores. Notaremos como $|T_t|$ a la cantidad de constructores presentes en el tipo de datos T_t, donde t es usado para indexar entre los distintos tipos de datos involucrados en el proceso de generación aleatoria.

TEOREMA 1: *Considere los generadores aleatorios derivados mediante nuestra herramienta para los tipos de datos (posiblemente) mutuamente recursivos $\{T_t\}_{t=1}^{k}$ con constructores $\{C_i^t\}_{i=1}^{|T_t|}$. Además considere las matrices de reproducción de constructores M_C y de reproducción de tipos M_T para los tipos de datos antes citados. Luego, sean G_n^C y G_n^T las n-ésimas generaciones de constructores y de placeholders de tipos respectivamente. Asumiendo $p_{C_i^t}$ como la probabilidad de generar el constructor $C_i^t \in T_t$, luego se satisface que:*

$$\left(E\left[G_n^C\right]\right).C_i^t = \left(E\left[G_n^T\right]\right).T_t \cdot p_{C_i^t} \tag{4.15}$$

■

Coloquialmente, el número esperado de constructores C_i^t en la generación n consiste del numero esperado de placeholders de su tipo de datos T_t en la generación n, multiplicado por la probabilidad de generar dicho constructor. Este teorema nos permite simplificar enormemente nuestros cálculos usando la matriz de reproducción de tipos en lugar de la matriz de reproducción de constructores, la cual siempre es de menor tamaño que esta última. Notar sin embargo que sigue siendo necesario considerar de manera separada los constructores terminales producidos en el último nivel del proceso de generación.

En esta sección extendemos nuestro mecanismo predictivo de manera modular con el fin de soportar la predicción de constructores para tipos de datos compuestos. Es decir, tipos de datos cuyos constructores incluyen argumentos de otros tipos de datos primitivos o previamente definidos pero que no se ven involucrados en el proceso de ramificación. Para ilustrar esta extensión, modificaremos el tipo de datos $Tree_{3L}$ previamente definido con el fin de almacenar datos booleanos en las hojas:

$$\texttt{data}\ Tree_{3L} = Leaf_A \mid Leaf_B\ Bool \mid Leaf_C\ Bool\ Bool \mid Node\ Tree_{3L}\ Tree_{3L}$$

Para predecir el número esperado de constructores *True* generados[***] , simplemente debemos calcular la distribución esperada de constructores de $Tree_{3L}$ mediante el mecanismo predictivo mostrado en las secciones anteriores, luego multiplicar el número esperado de cada constructor por el número de argumentos de tipo *Bool* presentes en cada una de ellos, y finalmente multiplicar este resultado por la probabilidad de generar un constructor *True*. Formalmente:

$$\begin{aligned} E[True] &= p_{True} \cdot (\overbrace{0 \cdot E[Leaf_A]}^{Leaf_A} + \overbrace{1 \cdot E[Leaf_B]}^{Leaf_B\ Bool} + \overbrace{2 \cdot E[Leaf_C]}^{Leaf_C\ Bool\ Bool} + \overbrace{0 \cdot E[Node]}^{Node\ Tree_{3L}\ Tree_{3L}}) \\ &= p_{True} \cdot (E[Leaf_B] + 2 \cdot E[Leaf_C]) \end{aligned}$$

En este caso, *Bool* es un tipo de datos primitivo y la predicción resulta directa. Sin embargo, las predicciones resultan más interesantes si deseamos usar tipos compuestos más complejos que involucren, por ejemplo, tipos de datos polimórficos. Para ilustrar este último punto, consideremos una versión levemente modificada de $Tree_{3L}$ donde incorporamos el tipo de datos polimórfico *Maybe a*:

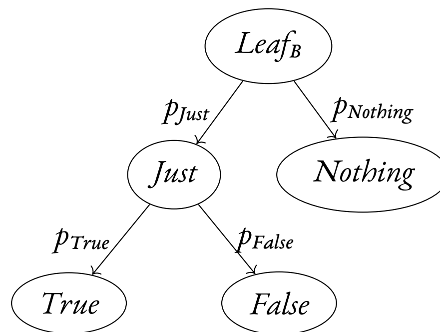
$$\begin{aligned} &\texttt{data}\ Maybe\ a = Nothing \mid Just\ a \\ &\texttt{data}\ Tree_{3L} = Leaf_A \mid Leaf_B\ (Maybe\ Bool) \mid Leaf_C\ Bool\ Bool \mid Node\ Tree_{3L}\ Tree_{3L} \end{aligned}$$

[***] De manera análoga podemos estimar el número esperado de constructores *False* generados.

En este caso, el constructor $Leaf_B$ posee un argumento de tipo *Maybe Bool*, para el cual puede generarse tanto un constructor *Nothing* como un constructor *Just* que almacena un valor booleano. Para predecir el número esperado de constructores *True* ahora además debemos considerar los casos en los que este valor de tipo *Maybe Bool realmente contiene un valor booleano*, es decir, cuando se genera un constructor *Just*. Este caso particular puede calcularse de la siguiente manera:

$$E[True] = p_{True} \cdot (p_{Just} \cdot 1 \cdot E[Leaf_B] + 2 \cdot E[Leaf_C])$$

En el caso general, donde se tienen constructores que poseen tipos de datos compuestos en varios niveles, es necesario conocer la "cadena" de constructores generados que dan lugar a la generación de un valor del tipo externo al proceso de ramificación que estamos buscando predecir. En el caso anterior, es necesario conocer que un constructor *True* puede ser generado si se genera un constructor $Leaf_B$ junto con un constructor *Just* entre ellos. Para obtener esta información, proponemos construir lo que llamamos un *grafo de dependencias de constructores*. Es decir, un grafo dirigido donde cada nodo representa un constructor, mientras que cada arista representa la posibilidad de que un constructor genere a otro. Además, las aristas de este grafo están etiquetadas con las probabilidades correspondientes a las dependencias antes mencionadas. A continuación se muestra el grafo de dependencias de constructores con el constructor $Leaf_B$ como nodo raíz.



Habiendo generado este grafo, y aplicando el modelo predictivo basado en procesos de ramificación ya explicado, podemos predecir fácilmente el número esperado de constructores pertenecientes a tipos de datos externos al proceso de ramificación. Es suficiente con multipli-

car las probabilidades presentes en cada arista del camino entre cada constructor involucrado en el proceso de ramificación y el constructor externo deseado.

El modelo predictivo basado en procesos de ramificación, junto con las extensiones a éste introducidas hasta aquí nos permiten predecir las distribuciones de constructores obtenidas mediante generadores aleatorios derivados usando nuestra herramienta para tipos de datos no triviales, los cuales puede encontrarse en muchas librerías de Haskell de uso masivo.

5
Evaluación

Hasta aquí hemos visto en detalle cómo predecir de manera analítica la distribución de constructores inducida por generadores de datos aleatorios, cuidadosamente construidos para diversos tipos de datos usando nuestra herramienta basada en procesos de ramificación. La mayor diferencia entre nuestro enfoque de generación de datos en comparación al de herramientas existentes como *derive* o *MegaDeTH* recae en la posibilidad de asignar diferentes probabilidades de generación a cada constructor en particular. Luego, dada una asignación de probabilidades a constructores, es posible predecir el comportamiento de un generador aleatorio derivado usando éstas como parámetros de entrada. En este trabajo proponemos el uso de un mecanismo automático que optimice dichas probabilidades con el fin de modificar la distribución de constructores resultante, de manera que la misma se ajuste a las necesidades particulares del usuario, las cuales pueden ser expresadas mediante un lenguaje de dominio específico mínimo.

En este capítulo se discute tanto la implementación de nuestra herramienta de derivación y optimización automática de generadores de datos aleatorios, como su aplicación a casos de estudios no triviales y que proveen evidencia no negligible en favor de nuestro enfoque respecto del estado del arte en materia de random testing automático.

5.1. Implementación

En esta sección presentamos los conceptos claves en la implementación de *DRaGen*, nuestra herramienta de derivación y optimización automática de generadores aleatorios. *DRaGen* es una herramienta escrita en Haskell, haciendo uso intensivo de *Template Haskell*, el mecanismo de meta-programación en Haskell presentado en el Capítulo 2. *DRaGen* implementa el modelo predictivo basado en procesos de ramificación y sus extensiones presentados en el capítulo anterior, junto con un optimizador de distribuciones de datos, encargado de calibrar las probabilidades involucradas en los generadores aleatorios derivados automáticamente. Este optimizador se encuentra parametrizado mediante *funciones de costo*, las cuales guían el proceso de optimización hacia distribuciones de constructores que pueden ser expresadas mediante un conjunto reducido de funciones primitivas. La siguiente figura ilustra el proceso completo de derivación y optimización de generadores de datos aleatorios efectuado por nuestra herramienta.

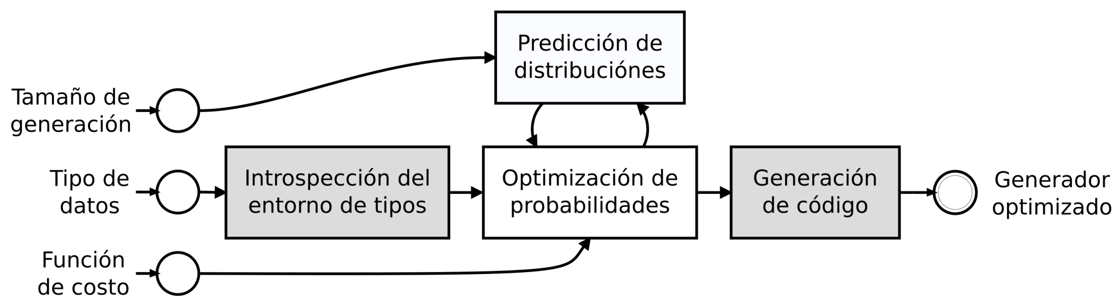


Figura 5.1: Proceso de derivación y optimización automática.

Las componentes sombreadas del proceso hacen uso de *Template Haskell* con el fin de interactuar con el entorno de compilación, mientras que el resto de ellas representa la etapa intermedia de optimización de probabilidades. Como se vio en el Capítulo anterior, este proceso puede ser invocado mediante la meta-función:

$$dragenArbitrary :: Name \rightarrow Size \rightarrow CostFunction \rightarrow Q\ [Dec]$$

Se puede observar que este proceso está parametrizado por i) el nombre del tipo de datos para el cual se desea obtener un generador optimizado, si se tratan de tipos mutuamente recursivos simplemente se indica el tipo raíz de la generación, ii) el tamaño de generación deseado, necesario para que nuestro mecanismo predictivo sea capaz de predecir la distribución de constructores en la última generación, y iii) una función de costo, la cual codifica la distribución de constructores deseada por el programador, retornando un número real que se busca minimizar. A partir de estos parámetros, nuestra herramienta se encargará del resto del proceso de derivación de manera completamente automática, insertando en tiempo de compilación un generador de datos aleatorios optimizado.

5.1.1. Proceso de derivación

Internamente, el proceso de derivación comienza efectuando una introspección del entorno de compilación, obteniendo información relevante respecto de los tipos de datos involucrados en el proceso de generación aleatoria, analizando las definiciones de los mismos en búsqueda de posibles tipos de datos mutuamente recursivos y tipos de datos compuestos.

La etapa intermedia del proceso de derivación consiste de un optimizador de probabilidades de generación, que hace uso de nuestro modelo predictivo basado en procesos de ramificación, parametrizado por la función de costo especificada. Este optimizador está basado en un algoritmo estándar de búsqueda local de vecinos recursiva. En este contexto, los vecinos representan distribuciones de probabilidad sobre los constructores, los cuales se construyen perturbando la probabilidad de cada constructor mediante un valor Δ predefinido. Este comportamiento puede observarse en el Algoritmo 1. El símbolo $\triangleright$ denota el inicio de un comentario. La función *neighborhood* parte de la distribución de probabilidad actual, la cual puede pensarse como un vector de probabilidades indexado por cada constructor involucrado en el proceso de derivación. Luego, se procede a iterar sobre cada constructor presente en esta distribución de probabilidades, generando por cada uno de ellos dos nuevos vecinos (*lower_neighbor* y *upper_neighbor*) perturbando *sólo ese constructor* por $\pm\Delta$. Luego, cada par de nuevos vecinos se agrega al conjunto vecindario de la distribución actual de probabilidades. Notar la necesidad de efectuar el chequeo de valores positivos y normalización, con el fin de asegurar que cada vecino representa una distribución de probabilidad—ver líneas 7 y 8 del Algoritmo 1.

Algoritmo 1 Calcula los vecinos inmediatos de una asignación de probabilidades.

```
1: function NEIGHBORHOOD(current_probs)
2:     neighbors ← ∅
3:     for each con in constructors do            ▷ Perturbar cada probabilidad por ∓Δ
4:         upper_neighbor ← current_probs
5:         lower_neighbor ← current_probs
6:         upper_neighbor[con] ← current_probs[con] + Δ
7:         lower_neighbor[con] ← max(0, current_probs[con] − Δ)
8:         neighbors ← neighbors ∪ {norm(upper_neighbor), norm(lower_neighbor)}
9:     end for
10:    return neighbors
11: end function
```

Con el fin de determinar la "mejor" distribución de probabilidades, el algoritmo de búsqueda local aplica nuestro modelo predictivo sobre todos los vecinos inmediatos de la distribución de probabilidades actual que no han sido todavía visitados, y luego evalúa la función de costo provista sobre cada uno de ellos en búsqueda del mejor candidato. Este proceso continúa recursivamente hasta que se encuentra un mínimo local—cuando no existen vecinos por evaluar—, o hasta que la diferencia de costo observada entre la distribución de probabilidades actual y el mejor vecino sea menor a un ε predeterminado. Este comportamiento se muestra en el Algoritmo 2.

Finalmente, la última etapa del proceso de derivación se encarga de sintetizar una instancia de la clase de tipos *Arbitrary* para el tipo de datos deseado y todos los tipos intermedios necesarios utilizando las probabilidades previamente optimizadas. Para esto, extendemos ciertas funcionalidades presentes en *MegaDeTH*. Una vez completada esta etapa, el resto del código de usuario puede generar valores aleatorios del tipo de datos deseado llamando a *arbitrary* como se mostró en el Capítulo 2.

Cabe destacar que el uso de un algoritmo de optimización en lugar de la búsqueda de una solución analítica es una decisión de diseño que se ve respaldada por dos aspectos importantes del problema que buscamos atacar. En primer lugar, el costo computacional de encontrar una solución exacta a un sistema no-lineal de ecuaciones (como aquellos presentes en los procesos de ramificación multi-tipo) puede ser prohibitivamente alto cuando se manipulan una gran

Algoritmo 2 Optimiza la distribución de probabilidades de constructores minimizando la función de costo provista.

```
1:  function OPTIMIZE(initial_probs, cost_function)
2:      best_probs ← initial_probs
3:      visited ← {initial_probs}
4:      repeat
5:          new_neighbors ← neighborhood(best_probs) \ visited    ▷ Expandimos la frontera
6:          step_delta ← 0
7:          for each neighbor in new_neighbors do
8:              if cost_function(neighbor) < cost_function(best_probs) then
9:                  step_delta ← cost_function(best_probs) − cost_function(neighbor)
10:                 best_probs ← neighbor                  ▷ Encontramos un mejor candidato
11:             end if
12:         end for
13:         visited ← visited ∪ new_neighbors
14:     until new_neighbors = ∅ ∨ step_delta < ε
15:     return best_probs
16: end function
```

cantidad de constructores, y por ende una gran cantidad de incógnitas a resolver. En segundo lugar, la existencia de dicha solución exacta no está garantizada en la mayoría de los casos, debido principalmente a los invariantes presentes en los tipos de datos manipulados. Por ejemplo, todo árbol binario con n nodos posee exactamente $n+1$ hojas, por lo que encontrar una solución que nos permita generar la misma cantidad tanto de nodos como de hojas es imposible. En dichos casos, en lugar de abortar completamente el proceso de optimización, creemos que es mucho más útil para el usuario obtener un generador aleatorio con una distribución de constructores que se aproxime suficientemente bien al objetivo deseado. De esta manera, ni el usuario ni el mecanismo de derivación automática deben estar al tanto de los invariantes que los tipos de datos que manipulan puedan poseer.

5.1.2. Funciones de costo

Como se introdujo antes, el proceso de optimización de nuestra herramienta es guiado mediante una función de costo provista por el usuario. Esta función asigna un número no-negativo—

el costo—a cada distribución de probabilidades de constructores que recibe como entrada. Se busca que cada función de costo retorne un número progresivamente más pequeño a medida que la predicción de la distribución de constructores (obtenida mediante la distribución de probabilidades de constructores de entrada) se acerque a cierta distribución de constructores objetivo, la cual depende de cada función de costo en particular. Luego, el proceso de optimización simplemente busca la distribución de probabilidades de constructores que minimice la salida de la función de costo provista. Actualmente, nuestra herramienta provee un conjunto mínimo de funciones de costo para describir distribuciones de constructores, el cual puede ser fácilmente extendido por el usuario. Estas funciones están construidas en torno al test estadístico conocido como *Chi-Square Goodness of Fit Test*[2], o simplemente X^2, el cual permite cuantificar en qué medida un fenómeno observado difiere de un valor esperado. El mismo viene dado por la siguiente fórmula:

$$X^2(\Gamma) = \sum_{C \in \Gamma} \frac{(observed(C) - expected(C))^2}{expected(C)} \tag{5.1}$$

En este contexto, Γ es un subconjunto de todos los constructores involucrados en el proceso de generación aleatoria. Por otro lado, $observed(C)$ viene dado por la predicción del número esperado de constructores C generados usando la distribución de probabilidades de entrada, mientras que $expected(C)$ es el número esperado de constructores C que se busca en la distribución de constructores del generador optimizado, que depende en particular de cada función de costo y parámetros que ésta posea. Este test estadístico fue elegido empíricamente de entre varios posibles tipos de tests estadísticos, ya que fue aquel con el que se consiguieron los mejores resultados en la mayoría de los casos estudiados. A continuación detallamos algunas funciones de costo provistas por nuestra herramienta.

Generación uniforme: si el usuario desea una distribución de constructores en la que el número esperado de cada uno de ellos sea lo más cercano posible, la función *uniform* :: *CostFunction* permite expresar esta restricción de manera sencilla. La misma funciona minimizando la diferencia entre el número esperado de constructores de cada constructor involucrado en el proceso de generación y el tamaño de generación. En otras palabras, se busca que $\forall C : expected(C) = n$, donde n es el tamaño de generación.

Generación restringida: similar a *uniform*, la función de costo $onlyCons :: [Name] \rightarrow CostFunction$ permite obtener una distribución uniforme de constructores, esta vez compuesta únicamente por aquellos constructores especificados (*whitelisting*). De manera complementaria, $withoutCons :: [Name] \rightarrow CostFunction$ permite obtener una distribución uniforme de constructores en la que no se genera ninguno de los constructores especificados (*blacklisting*). Adicionalmente, el usuario puede especificar distribuciones en las que se permite o se restringe la generación de constructores a nivel de tipos de datos, de manera que todo constructor perteneciente a alguno de los tipos listados es permitido o bloqueado—útil en presencia de tipos mutuamente recursivos. Las funciones $onlyTypes :: [Name] \rightarrow CostFunction$ y $withoutTypes :: [Name] \rightarrow CostFunction$ son las encargadas de proveer esta funcionalidad.

Generación ponderada: la última función de costo que presentamos es $weighted :: [(Name, Int)] \rightarrow CostFunction$, la cual permite especificar explícitamente una generación aleatoria donde el número esperado de ciertos constructores generados sigue una proporción dada. La probabilidad de generación de aquellos constructores no especificados explícitamente en la lista de "pesos" de entrada no contribuye al costo resultante de esta distribución de probabilidades. Esto permite que el optimizador modifique libremente las probabilidades de generación de estos constructores, con el fin de ajustar la distribución de los constructores explícitamente contemplados a las proporciones deseadas por el usuario. Cabe destacar que, la función de costo *uniform* puede pensarse como un caso particular de uso de *weighted*, donde todos los constructores involucrados en el proceso de generación son explícitamente especificados usando la misma proporción. El funcionamiento de esta función de costo en particular puede observarse en el Algoritmo 3.

Algoritmo 3 Función de costo para generación ponderada de constructores.

```
1:  function WEIGHTED(probs, cons_weights)
2:      prediction ← predict(probs)          ▷ Distribución esperada indexada por constructor
3:      cost ← 0
4:      for each (con, weight) in cons_weights do                          ▷ Calculamos χ²
5:          observed ← prediction[con]
6:          expected ← weight ∗ gen_size
7:          cost ← cost + (observed − expected)² / expected
8:      end for
9:      return cost
10: end function
```

Con el fin de ilustrar la forma de las posibles distribuciones de constructores obtenidas usando diferentes funciones de costo, la siguiente tabla muestra algunos ejemplos obtenidos mediante un generador aleatorio derivado automáticamente para el tipo $Tree_{3L}$ antes visto, y usando un tamaño de generación igual a 10.

Función de Costo	Número Esperado			
	$Leaf_A$	$Leaf_B$	$Leaf_C$	*Node*
uniform	5.26	5.26	5.26	14.73
weighted $[('Leaf_A, 3), ('Leaf_B, 1), ('Leaf_C, 1)$	29.97	9.99	9.99	48.96
weighted $[('Leaf_A, 1), ('Node, 3)]$	10.00	5.97	14.93	29.91
only $['Leaf_A, 'Node]$	9.50	0	0	10.50
without $['Leaf_C]$	6.94	7.04	0	12.99

En la tabla anterior pueden observarse algunos detalles interesantes. En el primer caso (*uniform*), el optimizador es incapaz de violar el invariante de $Tree_{3L}$, el cual dicta que la cantidad esperada de hojas siempre supera por uno a la cantidad esperada de nodos. En cambio, éste converge a una distribución que aproxima una generación de constructores “uniforme”, donde la cantidad esperada de cada constructor se encuentra alrededor del tamaño de generación. Como se comentó antes, creemos que este es un comportamiento deseable, ya que se obtiene una

solución suficientemente buena para la mayoría de los casos de uso práctico, y sin la necesidad de conocer los invariantes específicos del tipo de datos en cuestión. Por otro lado, en el segundo caso ($weighted\ [('Leaf_A, 3), ('Leaf_B, 1), ('Leaf_C, 1)]$), puede observarse que el número esperado de constructores *Node* es considerablemente alto. Esto se debe a que *Node* no fue explícitamente especificado entre los constructores para los cuales se requiere preservar cierta proporción, y por ende, el optimizador tiene libertad de ajustar su probabilidad con el fin de satisfacer el resto de las proporciones explicitadas para las hojas.

La siguiente sección evalúa el poder de *DRaGen*, derivando generadores de datos aleatorios para distintos tipos de datos no triviales, y comparando su capacidad de generación ante el resto de los mecanismos de derivación automática de generadores de datos aleatorios antes vistos.

5.2. Casos de Prueba

En esta sección ponemos a prueba nuestra herramienta de derivación y optimización automática de generadores presentada en la sección anterior ante casos de uso real.

Comenzamos comparando los generadores aleatorios para el tipo de datos $Tree_{3L}$ derivados automáticamente mediante *MegaDeTH* y *Feat*, presentados en el Capítulo 3, con el generador aleatorio correspondiente derivado mediante nuestra herramienta usando *uniform* como función de costo. Se usó un tamaño de generación igual a 10 tanto para el caso de *MegaDeTH* como para el caso de nuestra herramienta. Recordando que *Feat* utiliza una semántica distinta para interpretar el tamaño de generación, y con el fin de compararlo con los otros dos mecanismos de derivación automática, se decidió utilizar en este caso particular un tamaño de generación igual a 400, y por ende un máximo de 400 constructores, ya que éste fue el máximo número de constructores generados utilizando nuestra herramienta con el tamaño de generación citado arriba. La siguiente figura ilustra las distribuciones de los valores generados mediante los tres mecanismos, relacionando el número de valores generados respecto de la cantidad de constructores que conforman a cada uno de ellos.

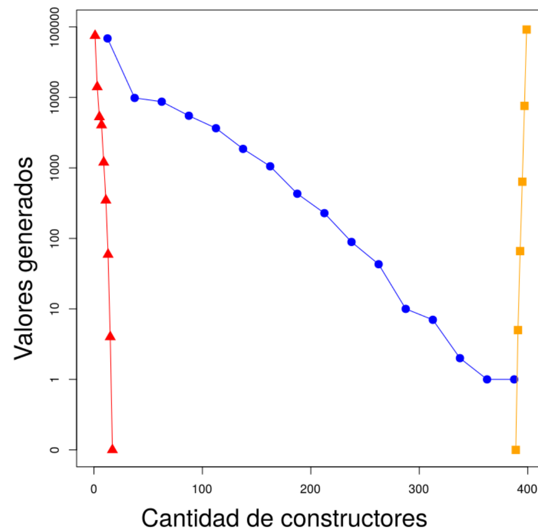


Figura 5.2: *MegaDeTH* (▲) vs. *Feat* (■) vs. *DRaGen* (●) usando el tipo de datos $Tree_{3L}$.

Como también se vio en las figuras 3.2 y 3.4 del Capítulo 3, los generadores derivados usando tanto *MegaDeTH* como *Feat* poseen una capacidad muy limitada de producir una amplia variedad de valores de diferentes tamaños. En contraste, el generador aleatorio optimizado derivado utilizando *DRaGen* es capaz de generar una gran variedad de valores de distintos tamaños, desde aquellos muy pequeños a algunos muy grandes.

Por otro lado, se espera que cuanto más variado sean los valores generados aleatoriamente, mayores sean las chances de cubrir una mayor superficie de código fuente cuando se testea software externo, contando con una cantidad de tiempo disponible limitada. Los casos de prueba mostrados a continuación proveen evidencia en esa dirección.

Con el fin de evaluar qué tan bien se comportan los generadores aleatorios obtenidos usando nuestra herramienta, se eligieron tres programas ampliamente utilizados y de distinta complejidad. Estas aplicaciones son I) *GNU bash 4.4*[4], la shell de Unix más utilizada, II) *GNU CLISP 2.49*[5], el compilador e intérprete del lenguaje de programación Common Lisp elegido por una gran cantidad de sistemas y III) *giffix*, una pequeña utilidad de software presente en la librería de manipulación de imágenes Gif *GIFLIB 5.1*[14]. Es importante remarcar que estas aplicaciones no están escritas en Haskell. No obstante, existen librerías de Haskell capaces inter-operar con estos programas mediante sus respectivos formatos de archivo: *language-*

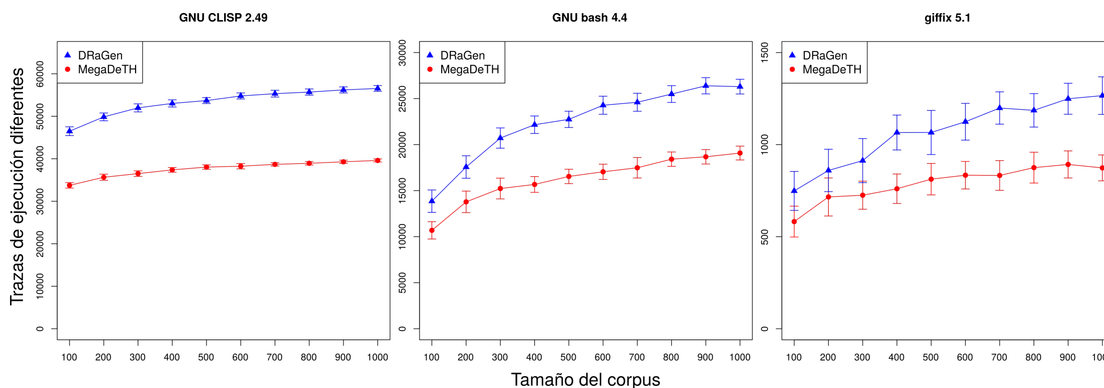
bash[23], *atto-lisp*[34] y *juicypixels*[35], respectivamente. Estas librerías proveen definiciones de tipos de datos que representan, en cada caso, la estructura del formato de archivo que manipula cada una de las aplicaciones nombradas arriba, las cuales fueron utilizadas para derivar generadores de datos aleatorios automáticamente. Además, estas librerías proveen funciones de serialización, que nos permiten transformar los valores de Haskell generados aleatoriamente en archivos reales que pueden ser usados como entrada de dichas aplicaciones. La siguiente tabla ofrece al lector una idea estimada de la envergadura de los tipos de datos utilizados en nuestros experimentos.

Caso de estudio	Tipo de datos generado	Tipos de datos involucrados	Constructores involucrados	Tipos compuestos	Tipos mut. rec.
Lisp	*Data.AttoLisp.Lisp*	7	14	Si	Si
Bash	*Language.Bash.Syntax.Command*	31	136	Si	Si
Gif	*Codec.Picture.Gif.GifFile*	16	30	Si	No

Para el propósito de nuestros experimentos, utilizamos una medida de cubrimiento de código fuente conocida como *traza de ejecución*, la cual es muy utilizada para cuantificar la efectividad del testing aleatorio, y en particular es utilizada por *American Fuzzy Lop*[27] (AFL), un fuzzer de software muy prolífero, que usaremos como herramienta auxiliar de nuestros experimentos. El proceso comienza con la *instrumentación* de los binarios a verificar utilizando las funcionalidades de compilación provistas por AFL, haciéndolos capaces de retornar el *camino* en el código fuente recorrido en cada ejecución del mismo. Una vez instrumentados los binarios a verificar, es posible utilizar AFL para obtener el número de trazas de ejecución distintas que son provocadas al ejecutar cada binario ante un conjunto de archivos de entrada generados aleatoriamente—también conocido como un *corpus*. Cabe destacar que sólo usamos AFL para cuantificar el cubrimiento de código producido por las distintas herramientas de derivación, y de ninguna manera para modificar aleatoriamente (fuzzear) los corpuses generados. En nuestros experimentos comparamos en qué medida diferentes generadores aleatorios derivados usando *MegaDeTH* y *DRaGen* provocan diferencias en el cubrimiento de código fuente de los programas antes mencionados, utilizando una serie de corpuses independientemente generados, cada uno de entre 100 y 1000 archivos aleatorios. Lamentablemente, *Feat* no resulta suficientemente robusto como para lograr manipular los tipos de datos utilizados

en nuestros experimentos para derivar generadores aleatorios, provocando errores de compilación debido a casos de entrada no considerados en su implementación.

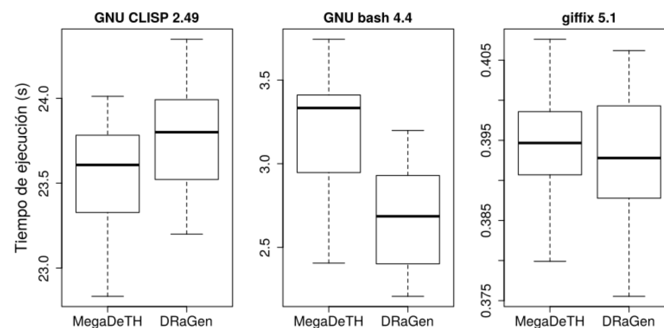
Luego, la generación aleatoria de cada corpus se realizó utilizando los mismos tipos de datos y tamaños de generación en cada caso. En particular, se utilizó un tamaño de generación igual a 10 para *Lisp* y *Bash*, e igual a 5 para el caso de *Gif*. Además, en el caso de *DRaGen* se utilizaron funciones de costo uniforme, buscando reducir cualquier posible bias externo. De esta manera, cualquier diferencia observable en el cubrimiento de código fuente obtenido usando cada una de las herramientas usadas es enteramente causada por la optimización de probabilidades de generación efectuada en la etapa intermedia de nuestro proceso de derivación automática, la cual no representa esfuerzo adicional para el usuario final más que elegir qué mecanismo de derivación utilizar. Por otro lado, con el fin de obtener resultados estadísticamente significativos, cada experimento fue repetido 30 veces usando corpuses independientemente generados para cada combinación de herramienta de derivación utilizada y tamaño de corpus. Las siguientes figuras muestran el número medio de trazas de ejecución distintas producidas por cada par de generadores y tamaño de corpus, utilizando barras de error indicando intervalos de confianza del 95 %:



A partir de las figuras anteriores, se puede observar cómo aquellos generadores derivados utilizando nuestro enfoque predictivo logran generar casos de prueba capaces de desencadenar un cubrimiento del código fuente mucho mayor que aquellos derivados usando *MegaDeTH*. En particular, estos resultados indican una mejora promedio de entre 35 % y 41 % con un error

estándar cercano al 0.35 % en el número de diferentes trazas de ejecución contabilizadas sobre los programas bajo prueba.

Un lector atento podría recordar que, como se mostró en la Figura 3.2, *MegaDeTH* posee una tendencia a derivar generadores aleatorios que producen casos de prueba muy pequeños. Si además consideramos que los casos de prueba pequeños deberían necesitar (en promedio) una menor cantidad de tiempo para ser evaluados que aquellos de mayor tamaño, entonces existe cierta justificación de creer que, teniendo el mismo tiempo disponible, evaluar un mayor número de casos de pruebas pequeños compensa su incapacidad de provocar un mayor cubrimiento de código fuente respecto de evaluar un menor número de casos de prueba de mayor tamaño. Sin embargo, cuando se testea software externo como en nuestros experimentos, es importante considerar el costo computacional introducido por el sistema operativo—debido mayormente al lanzamiento de procesos y cambios de contexto. Debido a esto, es usualmente preferible evaluar casos de prueba *más interesantes* en lugar de casos de prueba pequeños. La siguiente figura muestra los tiempos de ejecución requeridos por cada aplicación bajo prueba para evaluar todos los archivos presentes en los corpus antes generados de mayor tamaño, cada uno con 1000 casos de prueba:



Se puede observar que la diferencia de tamaño de los casos de prueba generados usando tanto *DRaGen* como *MegaDeTH* no produce diferencias considerables en los tiempos de ejecución requeridos para evaluar cada corpus. De esta manera, un usuario tiene mayores posibilidades de conseguir mejores resultados utilizando nuestra herramienta en lugar de *MegaDeTH*, con virtualmente *el mismo esfuerzo requerido*.

Por otro lado, creemos importante remarcar que, si se ejecuta un número suficientemente grande de casos de prueba (generando corpuses de mayor tamaño), luego se espera que el cubrimiento de código fuente tienda al 100 % del código alcanzable en ambos casos. Sin embargo, en la práctica, nuestra herramienta posee mejores posibilidades de alcanzar un mayor cubrimiento de código fuente al utilizar un número razonable de casos de prueba.

Los resultados anteriores nos aportan confianza para creer que nuestra herramienta, junto con el modelo predictivo subyacente, ofrecen una mejora substancial al estado del arte en materia de random testing automático en Haskell.

6
Conclusiones

En este trabajo se descubre una interacción entre la teoría de procesos estocásticos de ramificación y la generación aleatoria de valores siguiendo un enfoque basado en tipos de datos algebraicos, la cual no había sido estudiada hasta la fecha. Esta conexión nos permite formular una base matemática que describe el comportamiento observado en la distribución de constructores inducida por una variedad de generadores aleatorios cuidadosamente construidos usando *QuickCheck*. Usando este formalismo, diseñamos una heurística capaz de optimizar las probabilidades de generación de cada constructor, con el fin de ajustar el número esperado de cada constructor generado de acuerdo a las demandas del programador. Junto a esto, conectamos teoría y práctica mediante una herramienta de derivación automática de generadores aleatorios optimizados escrita en Haskell, cuya capacidad es puesta a prueba ante casos de estudio reales y de envergadura no trivial.

Sin embargo, este enfoque posee ciertas limitaciones teóricas y prácticas. En primer lugar, no se posee una noción de distribución de datos generados más allá de el número esperado de cada constructor sobre cada valor generado. Esto hace posible modelar ciertos invariantes tales como generación uniforme o ponderada de constructores, pero imposibilita considerar muchos invariantes estructurales interesantes, y que dan lugar a estructuras de datos com-

plejas tales como red-black trees, árboles de búsqueda o programas con scoping de términos. Algunas herramientas de testing aleatorio como *Luck*[24] o la presentada recientemente en [25] permiten derivar generadores que preservan invariantes definidos mediante relaciones inductivas, en especial dentro del marco de lenguajes de programación con sistemas de tipos dependientes. En este trabajo abordamos un objetivo mucho más modesto: diseñar un mecanismo que controle la generación de datos estructurados sin la necesidad de conocer a priori ningún invariante específico de los datos generados, una situación que ocurre comúnmente al realizar *testing de penetración*.

Otra limitación viene dada por el modelo de testing en sí. En general, no resulta deseable evaluar cada caso de prueba más de una única vez, puesto que esto degrada la performance del proceso de testing. *QuickCheck*, sin embargo, no provee herramientas que permitan evitar esto, ya que existe un costo computacional elevado relacionado a verificar que cada valor generado es único. De esta manera, la efectividad de nuestros generadores aleatorios depende de qué tan variados sean los valores que éstos generan, dado que no resulta práctico evaluar si ya han sido generados previamente o no. Para aquellos casos donde evaluar cada caso de prueba resulta sumamente costoso, es recomendable analizar si este enfoque de testing resulta efectivo, en contraste con otros modelos de testing que permiten generar casos de prueba sin repetición y de manera exhaustiva tales como *SmallCheck*[31].

Como se ha mostrado a lo largo de este trabajo, no existe un método universal en términos de random testing que garantice todas las propiedades deseables del proceso de testing de software. Sin embargo creemos que los aportes presentados en esta tesina proveen una mejora substancial al estado del arte en materia de testing aleatorio automático. Los mismos se pueden enumerar como sigue:

1. Se descubrió una relación entre los procesos estocásticos de ramificación y la generación aleatoria de datos utilizados al efectuar testing aleatorio.

2. Se desarrolló un modelo capaz de predecir la distribución de constructores inducida por generadores aleatorios derivados automáticamente a partir de la definición del tipo de datos deseado.

3. Se implementó una herramienta de derivación automática de generadores de datos alea-

torios, la cual es capaz de explotar el mecanismo predictivo antes desarrollado con el fin de optimizar las distribuciones de constructores respecto de las demandas del usuario.

4. Se evaluó nuestra herramienta ante casos de pruebas de considerable complejidad, obteniendo resultados favorables respecto de aquellos obtenidos mediante *MegaDeTH*.

A continuación se muestran algunas aristas de esta tesina que pueden dar lugar a trabajo futuro relacionado.

6.1. Trabajo Futuro

Existen diversas extensiones posibles tanto al modelo matemático como a la herramienta de derivación automática de generadores presentados en este trabajo.

En primer lugar, buscamos extender la noción de distribución de constructores a un modelo que nos permita caracterizar *patrones de constructores* generados, por ejemplo (*Node Leaf x*). De tal manera, podrían extraerse automáticamente los patrones utilizados por las funciones definidas en el nivel principal de un módulo Haskell, y luego derivarse generadores que ejerciten estos patrones de manera uniforme.

Otra extensión interesante sobre el modelo matemático presentado sería la capacidad de utilizar probabilidades de generación de constructores no estáticas, sino que en cambio las mismas pudiesen ser parametrizadas mediante, por ejemplo, el nivel de generación en el cual se está generando cada constructor. De esta manera, la probabilidad p_C de generar el constructor C se transformaría en una función $p_C :: GenLevel \rightarrow Double$ dependiente del nivel en el cual se encuentra el proceso de generación.

Por otro lado, existen otros escenarios donde los tipos de datos manipulados prácticamente no poseen estructura. Este el caso general de aquellos lenguajes de dominio específico definidos mediante *shallow embedding*, los cuales cargan el peso de preservar los invariantes del tipo de datos en cuestión sobre las funciones primitivas del lenguaje. Algunos ejemplos reales de esta práctica pueden encontrarse en librerías de Haskell como *blaze-html*, *language-css* o *ansi-terminal*. En estos casos, puede utilizarse un enfoque como el presentado en [10, 18] para derivar previamente un tipo de datos de alto orden que codifique las primitivas de dicho

lenguaje de dominio específico en la forma de un “álgebra de la librería”, para luego aplicar nuestro enfoque predictivo generando valores aleatorios de este nuevo tipo de datos, los cuales son luego traducidos a valores del tipo de datos deseado de manera automática.

A

Demostraciones

En este apéndice se provee el desarrollo formal requerido para mostrar que la matriz de reproducción de tipos puede ser usada para predecir la distribución de constructores. Comenzaremos definiendo cierta terminología.

Definición 1: *Sea T_t un tipo de datos definido como la suma de constructores:*

$$T_t := C_1^t + C_2^t + \cdots + C_n^t$$

Donde cada constructor a su vez se define como el producto de los tipos de datos:

$$C_c^t := T_1 \times T_2 \times \cdots \times T_m$$

Definiremos las siguientes funciones de observación:

$$cons(T_t) = \{C_c^t\}_{c=1}^n \qquad args(C_c^t) = \{T_j\}_{j=1}^m \qquad |T_t| = |cons(T_t)| = n$$

Además, definiremos el factor de ramificación de C_i^u a T_v *como el número* $\beta(T_v, C_i^u) \in \mathrm{N}$ *que denota la cantidad de ocurrencias de* T_v *en los argumentos de* C_i^u*:*

$$\beta(T_v, C_i^u) = |\{T_k \in args(C_i^u) \mid T_k = T_v\}|$$

Antes de presentar nuestro teorema principal, hace falta presentar algunas proposiciones preliminares. La siguiente proposición relaciona la media de reproducción de constructores con sus respectivos tipos de datos y el factor de ramificación.

PROPOSICIÓN 1: *Sea* M_C *la matriz de reproducción de constructores de un conjunto de tipos de datos (posiblemente) mutuamente recursivos* $\{T_t\}_{t=1}^{n}$ *con constructores* $\{C_i^t\}_{i=1}^{|T_t|}$*. Asumiendo* $p_{C_i^t}$ *como la probabilidad de generar un constructor* $C_i^t \in cons(T_t)$ *siempre que se necesite generar un valor de tipo* T_t*. Luego, se satisface que:*

$$M_C(i,j) = m_{C_i^u C_j^v} = \beta(T_v, C_i^u) \cdot p_{C_j^v} \tag{A.1}$$

Demostración. Sea $m_{C_i^u C_j^v}$ un elemento de M_C, sabemos que $m_{C_i^u C_j^v}$ representa el número esperado de constructores $C_j^v \in cons(T_v)$ generados siempre que un constructor $C_i^u \in cons(T_u)$ es generado. Puesto que cada constructor se compone del producto de (posiblemente) varios argumentos, necesitamos sumar el número esperado de constructores C_j^v generados por cada argumento de C_i^u de tipo T_v—el número esperado de constructores C_j^v generados por un argumento de tipo diferente a T_v es 0. Para esto, definimos la siguiente variable aleatoria $X_k^{C_i^u C_j^v}$ que captura el número de constructores C_j^v generados por el k-ésimo argumento de C_i^u como se muestra a continuación:

$$X_k^{C_i^u C_j^v} : cons(T_v) \to \mathbb{N}$$
$$X_k^{C_i^u C_j^v}(C_c^v) = \begin{cases} 1 & \text{si } c = j \\ 0 & \text{en otro caso} \end{cases}$$

Luego es posible calcular las probabilidades de generar cero o un constructores C_j^v por el k-ésimo argumento de C_i^u como sigue:

$$P\left(X_k^{C_i^u C_j^v} = 0\right) = 1 - p_{C_j^v}$$
$$P\left(X_k^{C_i^u C_j^v} = 1\right) = p_{C_j^v}$$

Y a partir de las probabilidades anteriores, podemos calcular la esperanza de cada $X_k^{C_i^u C_j^v}$:

$$E\left[X_k^{C_i^u C_j^v}\right] \;=\; 1 \cdot P\left(X_k^{C_i^u C_j^v} = 1\right) + 0 \cdot P\left(X_k^{C_i^u C_j^v} = 0\right) \;=\; p_{C_j^v} \tag{A.2}$$

Finalmente, podemos calcular el número esperado de constructores C_j^v generados siempre que se genera un constructor C_i^u sumando el número esperado de constructores C_j^v generados por cada argumento C_i^u de tipo T_v:

$$\begin{aligned}
m_{C_i^u C_j^v} &= \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} E\left[X_k^{C_i^u C_j^v}\right] \\
&= \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} p_{C_j^v} && \text{(por (A.2))} \\
&= p_{C_j^v} \cdot \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} 1 && (p_{C_j^v} \text{ es constante}) \\
&= p_{C_j^v} \cdot \left|\{T_k \in args(C_j^v) \mid T_k = T_v\}\right| && \left(\textstyle\sum_S 1 = |S|\right) \\
&= p_{C_j^v} \cdot \beta(T_v, C_i^u) && \text{(por def. de } \beta)
\end{aligned}$$

■

La siguiente proposición relaciona la media de reproducción de placeholders de tipos con sus respectivos constructores.

Proposición 2: *Sea M_T la matriz de reproducción de tipos de un conjunto de tipos de datos (posiblemente) mutuamente recursivos $\{T_t\}_{t=1}^{n}$ con constructores $\{C_i^t\}_{i=1}^{|T_t|}$. Asumiendo $p_{C_i^t}$ como la probabilidad de generar un constructor $C_i^t \in cons(T_t)$ siempre que se necesite generar un valor de tipo T_t. Luego, se satisface que:*

$$\mathcal{M}_T(u,v) = m_{T_u T_v} = \sum_{C_k^u \in cons(T^u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} \tag{A.3}$$

Demostración. Sea $m_{T_u T_v}$ un elemento de $\mathcal{M}_T$, sabemos que $m_{T_u T_v}$ representa el número esperado de placeholders de tipo T_v generados siempre que un placeholder de tipo T_u es generado (mediante cualquiera de los constructores de T_u). Por lo tanto, debemos contar el número promedio de placeholders de tipo T_v que aparecen en cada constructor de T_u. Para esto, introducimos la variable aleatoria Y^{uv} que captura este comportamiento.

$$Y^{uv} : cons(T_u) \to \mathbb{N}$$
$$Y^{uv}(C_k^u) = \beta(T_v, C_k^u)$$

Luego podemos obtener $m_{T_u T_v}$ calculando la esperanza de cada Y^{uv} como sigue:

$$\begin{aligned} m_{T_u T_v} &= E[Y^{uv}] \\ &= \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot P(Y^{uv} = C_k^u) && \text{(por def. de } E[Y^{uv}]) \\ &= \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} && \text{(por def. de } p_{C_k^u}) \end{aligned}$$

∎

La siguiente proposición relaciona cada entrada de la matriz $\mathcal{M}_T$ con sus correspondientes en la matriz $\mathcal{M}_C$.

PROPOSICIÓN 3: *Sean $\mathcal{M}_C$ y $\mathcal{M}_T$ respectivamente las matrices de reproducción de constructores y de tipos, de un conjunto de tipos de datos (posiblemente) mutuamente recursivos $\{T_t\}_{t=1}^{n}$ con constructores $\{C_i^t\}_{i=1}^{|T_t|}$. Asumiendo $p_{C_i^t}$ como la probabilidad de generar un constructor $C_i^t \in cons(T_t)$ siempre que se necesite generar un valor de tipo T_t. Luego, se satisface:*

$$p_{C_i^v} \cdot m_{T_u T_v} = \sum_{C_j^u \in cons(T_u)} m_{C_j^u C_i^v} \cdot p_{C_j^u} \tag{A.4}$$

Demostración. Sean C_i^u y C_j^v constructores pertenecientes a los tipos T^u y T^v, respectivamente. Luego, por (A.1) y (A.3) tenemos que:

$$m_{C_i^u C_j^v} = \beta(T_v, C_i^u) \cdot p_{C_j^v} \tag{A.5}$$

$$m_{T_u T_v} = \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} \tag{A.6}$$

Y reescribiendo (A.5) obtenemos:

$$\beta(T_v, C_i^u) = \frac{m_{C_i^u C_j^v}}{p_{C_j^v}} \qquad (\text{si } p_{C_j^v} \neq 0) \tag{A.7}$$

En el caso de que $p_{C_j^v} = 0$, la última ecuación se satisface trivialmente por (A.5). Luego, reemplazando (A.7) en (A.6) se obtiene:

$$m_{T_u T_v} = \sum_{C_k^u \in cons(T_u)} \frac{m_{C_i^u C_j^v}}{p_{C_j^v}} \cdot p_{C_k^u}$$

$$m_{T_u T_v} = \frac{1}{p_{C_j^v}} \cdot \sum_{C_k^u \in cons(T_u)} m_{C_i^u C_j^v} \cdot p_{C_k^u} \qquad (p_{C_j^v} \text{ es constante})$$

$$p_{C_j^v} \cdot m_{T_u T_v} = \sum_{C_k^u \in cons(T_u)} m_{C_i^u C_j^v} \cdot p_{C_k^u}$$

■

Finalmente, estamos en condiciones de probar nuestro resultado principal.

TEOREMA 1: *Considere los generadores aleatorios derivados mediante nuestra herramienta para los tipos de datos (posiblemente) mutuamente recursivos $\{T_t\}_{t=1}^{k}$ con constructores $\{C_i^t\}_{i=1}^{|T_t|}$. Además considere las matrices de reproducción de constructores M_C y de reproducción de tipos M_T para los tipos de datos antes citados. Luego, sean G_n^C y G_n^T las n-ésimas generaciones de constructores y de placeholders de tipos respectivamente. Asumiendo $p_{C_i^t}$ como la probabilidad de generar el constructor $C_i^t \in T_t$, luego se satisface que:*

$$\left(E\left[G_n^C\right]\right).C_i^t = \left(E\left[G_n^T\right]\right).T_t \cdot p_{C_i^t} \tag{A.8}$$

Demostración. Llamaremos T_r con $(1 \leq r \leq k)$ al tipo de datos raíz de la generación.

Luego, siguiendo el formalismo presentado en el Capítulo 4, se puede observar que las probabilidades iniciales $(E[G_0^C]).C_i^t$ y $(E[G_0^T]).T_t$ resultan como se muestra a continuación.

$$\left(E\left[G_0^C\right]\right).C_i^t = \begin{cases} p_{C_i^t} & \text{si } t = r \\ 0 & \text{en otro caso} \end{cases} \qquad \left(E\left[G_0^T\right]\right).T_t = \begin{cases} 1 & \text{si } t = r \\ 0 & \text{en otro caso} \end{cases}$$

Procedemos a demostrar nuestro resultado por inducción en n:

<u>Caso base</u>: queremos probar que $(E[G_0^C]).C_i^t = (E[G_0^T]).T_t \cdot p_{C_i^t}$. Sea T_t un tipo de datos involucrado en el proceso de ramificación.

- Si $T_t = T_r$, luego por las definiciones de probabilidades iniciales mostradas arriba tenemos que:

  $$\left(E\left[G_0^C\right]\right).C_i^t = p_{C_i^t} \qquad \left(E\left[G_0^T\right]\right).T_t = 1$$

  Y el teorema se satisface trivialmente.

- Si $T_t \neq T_r$, luego por las definiciones de probabilidades iniciales mostradas arriba tenemos que:

  $$\left(E\left[G_0^C\right]\right).C_i^t = 0 \qquad \left(E\left[G_0^T\right]\right).T_t = 0$$

  Y el teorema nuevamente se satisface de manera trivial.

<u>Caso inductivo</u>: queremos probar que $(E[G_n^C]).C_i^t = (E[G_n^T]).T_t \cdot p_{C_i^t}$. Por simplicidad, llamaremos $\Gamma = \{T_t\}_{t=1}^k$. Luego:

$$\begin{aligned} &\left(E\left[G_n^C\right]\right).C_i^t \\ &= E\left[\sum_{T_k \in \Gamma} \left(\sum_{C_j^k \in cons(T_k)} \left(G_{(n-1)}^C\right).C_j^k \cdot m_{C_j^k C_i^t} \right)\right] \qquad \text{(por def. proc. de ramificación)} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{T_k \in \Gamma} E\left[\sum_{C_j^k \in cons(T_k)} \left(G_{(n-1)}^{C} \right) . C_j^k \cdot m_{C_j^k C_i^t} \right] && \text{(por probabilidad)} \\
&= \sum_{T_k \in \Gamma} \left(\sum_{C_j^k \in cons(T_k)} E\left[\left(G_{(n-1)}^{C} \right) . C_j^k \cdot m_{C_j^k C_i^t} \right] \right) && \text{(por probabilidad)} \\
&= \sum_{T_k \in \Gamma} \left(\sum_{C_j^k \in cons(T_k)} E\left[\left(G_{(n-1)}^{C} \right) . C_j^k \right] \cdot m_{C_j^k C_i^t} \right) && \text{(por probabilidad)} \\
&= \sum_{T_k \in \Gamma} \left(\sum_{C_j^k \in cons(T_k)} \left(E\left[G_{(n-1)}^{C} \right] \right) . C_j^k \cdot m_{C_j^k C_i^t} \right) && \text{(por álgebra lineal)} \\
&= \sum_{T_k \in \Gamma} \left(\sum_{C_j^k \in cons(T_k)} \left(E\left[G_{(n-1)}^{T} \right] \right) . T_t \cdot p_{C_j^k} \cdot m_{C_j^k C_i^t} \right) && \text{(H.I.)} \\
&= \sum_{T_k \in \Gamma} \left(E\left[G_{(n-1)}^{T} \right] \right) . T_t \cdot \sum_{C_j^k \in cons(T_k)} p_{C_j^k} \cdot m_{C_j^k C_i^t} && \text{(por álgebra lineal)} \\
&= \sum_{T_k \in \Gamma} \left(E\left[G_{(n-1)}^{T} \right] \right) . T_t \cdot p_{C_i^t} \cdot m_{T_k T_t} && \text{(por (A.4))} \\
&= \sum_{T_k \in \Gamma} \left(E\left[G_{(n-1)}^{T} \right] \right) . T_t \cdot m_{T_k T_t} \cdot p_{C_i^t} && \text{(reordenamiento)} \\
&= \sum_{T_k \in \Gamma} E\left[\left(G_{(n-1)}^{T} \right) . T_t \right] \cdot m_{T_k T_t} \cdot p_{C_i^k} && \text{(por álgebra lineal)} \\
&= \sum_{T_k \in \Gamma} E\left[\left(G_{(n-1)}^{T} \right) . T_t \cdot m_{T_k T_t} \right] \cdot p_{C_i^t} && \text{(por probabilidad)} \\
&= E\left[\sum_{T_k \in \Gamma} \left(G_{(n-1)}^{T} \right) . T_t \cdot m_{T_k T_t} \right] \cdot p_{C_i^t} && \text{(por probabilidad)} \\
&= \left(E\left[G_n^{T} \right] \right) . T_t \cdot p_{C_i^t} && \text{(por def. proc. de ramificación.)}
\end{aligned}
$$

∎

Bibliografía

[1] Arts, T., Hughes, J., Norell, U., & Svensson, H. (2015). Testing AUTOSAR software with QuickCheck. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015* (pp. 1–4).

[2] Balakrishnan, N., Voinov, V., & Nikulin, M. (2013). *Chi-Squared Goodness of Fit Tests with Applications. 1st Edition*. Academic Press.

[3] Braquehais, R. & Runciman, C. (2016). Fitspec: refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016* (pp. 1–12).

[4] Brian J. Fox (1989). GNU Bash: The GNU Bourne-Again SHell. `https://www.gnu.org/software/bash/`.

[5] Bruno Haible (1994). GNU CLISP: an ANSI implementation of Common Lisp. `https://clisp.sourceforge.io/`.

[6] Bryan O' Sullivan (2011). aeson: Fast JSON parsing and encoding. `https://hackage.haskell.org/package/aeson`.

[7] CACA Labs (2010). zzuf - multi-purpose fuzzer. `http://caca.zoy.org/wiki/zzuf`.

[8] Cai, X. & Lyu, M. R. (2005). The effect of code coverage on fault detection under different testing profiles. *SIGSOFT Softw. Eng. Notes*, 30(4), 1–7.

[9] Claessen, K. & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Con-*

ference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.

[10] Claessen, K. & Hughes, J. (2002). Testing monadic code with QuickCheck. *SIGPLAN Not.*, 37(12), 47–59.

[11] Claessen, K., Smallbone, N., & Hughes, J. (2010). QuickSpec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs, TAP 2010, Málaga, Spain, July 1-2, 2010.* (pp. 6–21).

[12] Duregård, J., Jansson, P., & Wang, M. (2012). Feat: Functional enumeration of algebraic types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012* (pp. 61–72).

[13] Edward A. Kmett (2012). lens: Lenses, Folds and Traversals. `https://hackage.haskell.org/package/lens`.

[14] Eric S. Raymond (1989). GIFLIB: A library and utilities for processing GIFs. `http://giflib.sourceforge.net/`.

[15] Godefroid, P. (2007). Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07 (pp. 1–1). New York, NY, USA: ACM.

[16] Godefroid, P., Klarlund, N., & Sen, K. (2005). Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6), 213–223.

[17] Grieco, G., Ceresa, M., & Buiras, P. (2016). QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016* (pp. 13–20).

[18] Grieco, G., Ceresa, M., Mista, A., & Buiras, P. (2017). QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134(Supplement C), 340 – 354.

[19] Haccou, P., Jagers, P., & Vatutin, V. (2005). *Branching processes. Variation, growth, and extinction of populations.* Cambridge University Press.

[20] Hamlet, R. (1994). Random testing. *Encyclopedia of software Engineering*.

[21] Hudak, P. (1997). Domain-specific languages. *Handbook of programming languages*, 3(39-60), 21.

[22] Hughes, J., Norell, U., Smallbone, N., & Arts, T. (2016). Find more bugs with QuickCheck! In *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016* (pp. 71–77).

[23] Kyle Raftogianis (2017). language-bash: A library for parsing, pretty-printing, and manipulating Bash shell scripts. https://hackage.haskell.org/package/language-bash.

[24] Lampropoulos, L., Gallois-Wong, D., Hritcu, C., Hughes, J., Pierce, B. C., & Xia, L. (2017a). Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (pp. 114–129).

[25] Lampropoulos, L., Paraskevopoulou, Z., & Pierce, B. C. (2017b). Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), 45:1–45:30.

[26] Marlow, S. et al. (2010). Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011).*

[27] Michal Zalewski (2010). American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/.

[28] Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12), 32–44.

[29] Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93(1), 55–92.

[30] Neil Mitchell (2006). Data.derive: a library and a tool for deriving instances for haskell programs. https://hackage.haskell.org/package/derive.

[31] Runciman, C., Naylor, M., & Lindblad, F. (2008). Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIG-*

PLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008 (pp. 37–48).

[32] Sheard, T. & Jones, S. L. P. (2002). Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12), 60–75.

[33] Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[34] Thomas Schilling (2015). atto-lisp: Efficient parsing and serialisation of S-Expressions. https://hackage.haskell.org/package/atto-lisp.

[35] Vincent Berthoux (2012). Juicy.Pixels: Haskell library to load & save pictures. https://hackage.haskell.org/package/JuicyPixels.

[36] Wadler, P. & Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89 (pp. 60–76).

[37] Watson, H. W. & Galton, F. (1875). On the probability of the extinction of families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4, 138–144.