Universidad Nacional Autónoma de México

Facultad de Ciencias

Tipos anidados para estructuras cíclicas puramente funcionales

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
ALEJANDRO EHECATL MORALES HUITRÓN

TUTOR:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2014

Índice general

Introducción

La importancia del estudio de los fundamentos de los lenguajes de programación se debe a que nos da un marco de referencia para diseñar nuevos lenguajes o mejorar los existentes. A través de las propiedades sintácticas, operacionales o semánticas[21] se piensa en las condiciones que tiene un programador para construir sus programas de acuerdo al paradigma o los paradigmas involucrados. Muchos de estos conceptos están relacionados con teorías matemáticas abstractas como las categorías[23], o bien, sustentados en formalismos matemáticos como el cálculo lambda.

El paradigma de programación funcional se ha vuelto popular en los últimos años, su característica de tener sistemas fuertemente tipados, ya sea explícitamente o con inferencia de tipos, ha llevado al desarrollo de conceptos que además de tener interés por sí mismos, nos permiten definir estructuras de datos que en otro contexto son poco prácticas de manejar; un ejemplo de lo anterior son las estructuras cíclicas, es decir, estructuras que tienen apuntadores (o puntos de retorno) a ellas mismas, como por ejemplo los árboles cíclicos, presentes frecuentemente en ciencias de la computación (ver [11]). En estas estructuras, dada su naturaleza infinita, no es factible hacer inducción estructural (método usado para demostrar propiedades sobre las estructuras de datos) lo cual es una gran desventaja, ya que este método se utiliza para verificar propiedades de los programas.

Los tipos de datos anidados, llamados también tipos no regulares, surgen en una línea de investigación referente a la matemática de construcción de programas como constructores inductivos de segundo orden. Básicamente un tipo de dato anidado es un tipo de dato parametrizado, es decir, con un parámetro de tipo, el cual cambia del otro lado de la igualdad, a diferencia de lo que sucede con los tipos parametrizados recursivos, en los que el parámetro no cambia. Lo anterior dificulta la programación de funciones recursivas en lenguajes como Haskell o ML. Sin embargo, son conceptualmente importantes y útiles cuando son usados para diseñar estructuras de datos funcionales.

El propósito general de ese trabajo es implementar estructuras cíclicas puramente funcionales en Haskell mediante el uso de tipos anidados, haciendo una comparación con respecto a las implementaciones con tipos de datos regulares. También se dará un panorama de los conceptos fundamentales de las estructu-

ras de datos funcionales como son la persistencia y la programación mediante operadores de plegado, es decir, las funciones `foldr` y `unfoldr`.

En el capítulo uno se presenta un panorama del paradigma de programación funcional y las propiedades que lo distinguen de otros, el lenguaje de programación Haskell con sus tipos de datos básicos, los operadores de plegado como concepto central en la programación funcional, sus propiedades básicas y ejemplos de uso. En el capítulo dos se muestran estructuras infinitas y se dan ejemplos, también las estructuras cíclicas defindas con tipos de datos simples. En el capítulo tres se presentan las listas y árboles cíclicos definidos con tipos anidados, también se definen y discuten funciones de interés inmediato sobre esas estructuras. El posible trabajo a futuro con estos tipos de datos se encuentra en las conclusiones.

En este trabajo se utiliza el lenguaje de programación Haskell para los ejemplos y la implementación de las estructuras cíclicas con tipos anidados, ya que requieren el uso de funciones de orden superior.

Capítulo 1

Programación Funcional

Este trabajo se desarrolla dentro del *paradigma de Programación Funcional*, que es un estilo de programación que consiste en la definición de funciones y la aplicación de las mismas a argumentos como método básico de cómputo. Un lenguaje de programación funcional es aquel que soporta y promueve ese estilo. En contraste, el método básico de la programación imperativa es la asignación de variables.

En programación funcional, las funciones son entidades de primera clase, lo que indica que son tratadas como cualquier otro valor, pueden ser pasadas como argumentos en otras funciones y devolverse como resultados. También figuran en estructuras de datos[28].

1.1. Características de los lenguajes funcionales

1.1.1. Funciones de orden superior

Las *funciones de orden superior* son funciones que esperan a otras funciones como argumentos, regresan una función como resultado o tienen ambas características[15].

El ejemplo más común es la función *map* que toma una función y una lista, aplica la función a todos sus elementos y devuelve una nueva lista con los resultados.

```
map :: (a -> b) -> [a] -> [b]
map f [ ]      = [ ]
map f (x:xs)  = f x : map f xs
```

En la definición anterior se utilizan patrones del lado izquierdo de las igualdades. HASKELL reconoce patrones en estructuras de datos y los maneja por un

procedimiento conocido como ajuste de patrones (*pattern matching*).

El ajuste de patrones consiste en una especificación de pautas que deben ser seguidas por los datos, los cuales pueden ser deconstruidos permitiéndonos acceder a sus componentes. Los patrones en una función son verificados de arriba a abajo y cuando un patrón concuerda con el valor asociado, se evalúa la función en ese caso. Podemos usar el ajuste de patrones con cualquier tipo de dato.

En la función `map`, `[ ]` es un patrón que indica la lista vacía y `(x:xs)` una lista con cabeza `x` y cola `xs`[30]. De este modo el lenguaje permite hacer recursión estructural en listas:

- Se establece el caso base de la recursión con el patrón de lista vacía: dado que no tiene elementos, se regresa la lista vacía.

- En otro caso necesariamente se recibirá una lista con cabeza y cola (por el ajuste de patrones), y entonces se regresará una lista cuya cabeza es la aplicación de la función argumento a la cabeza de la lista original y cuya cola es la llamada recursiva de `map` con la función argumento y la cola de la lista original.

Ejemplos de funciones que utilizan `map`:

- ```
  mapDouble ::  [Int] -> [Int]
  mapDouble l = map double l
  ```

  La función `double` duplica el valor de un entero y a su vez es parámetro de `mapDouble` que realiza la misma acción a los elementos de una lista. Un ejemplo de evaluación (en una terminal de comandos del sistema operativo Linux en el cual se ejecuta el intérprete HUGS[1]) es el siguiente:

  ```
  > mapDouble [3,6,9,12]
     [6,12,18,24]
  ```

- ```
  mapFirst :: [(Int,Int)] -> [Int]
  mapFirst l = map fst l
  ```

  La función `fst` obtiene el primer parámetro de un par de datos (en HASKELL se llama tupla), a su vez la función `mapFirst` regresa los primeros elementos de una lista de pares.

  ```
  > mapFirst [(4,3),(3,2),(5,4),(7,9)]
     [4,3,5,7]
  ```

[1]De aquí en adelante se marcarán con el símbolo "`>`" las pruebas con el intérprete HUGS

1.1.2. Pureza

En programación funcional el valor de retorno es el resultado más importante en la evaluación de una función. Algunos lenguajes funcionales permiten expresiones que producen *acciones* además de devolver valores. Estas acciones se denominan *efectos laterales*. Los lenguajes que prohíben efectos laterales se les llama *puros* [28].

Los modelos de ejecución de lenguajes como ML, C o Java incluyen efectos laterales de control o almacenamiento. Los efectos de control causan una transferencia no local del control del programa, por ejemplo, lanzar una excepción, hacer un salto en el programa o el uso de continuaciones. Por otra parte, los efectos de almacenamiento son modificaciones dinámicas a la memoria mutable de un programa a través de asignación de variables[27].

La mutación del valor de una variable es una característica estandar en la mayoría de lenguajes de programación, incluso en lenguajes puros como Haskell existen mecanismos para manejar estados y asignaciones que requieren de conceptos avanzados de programación como las mónadas.

Por lo general es beneficioso escribir parte de un programa funcional de una manera puramente funcional, para así mantener el código que involucra cambios de estado y entrada/salida al mínimo, ya que como código impuro es más propenso a errores.

Datos inmutables

Los programas puramente funcionales suelen operar con datos inmutables. En lugar de modificar los valores existentes, se crean copias alteradas y el original se conserva. Dado que las partes sin cambios en la estructura no pueden ser modificadas, a menudo pueden ser compartidas entre las copias antiguas y las nuevas, lo que ahorra memoria.

Transparencia referencial

Los cómputos puros producen el mismo valor cada vez que son invocados. Esa propiedad se llama transparencia referencial y usualmente describe un estilo de programación en el que "iguales pueden ser reemplazados por iguales", lo que permite llevar a cabo el razonamiento ecuacional con el código[15]. Por ejemplo si

```
y = f x        y        g = h y y
```

entonces se puede sustituir a `g` por

```
g = h (f x) (f x)
```

y obtener el mismo resultado, sólo la eficiencia puede cambiar.

1.1.3. Recursión

Las funciones que se invocan a ellas mismas son llamadas recursivas. La recursión es ampliamente usada en programación funcional ya que es a menudo la única forma de iterar. Implementaciones de lenguajes funcionales generalmente optimizan con recursión de cola para asegurar que el proceso recursivo no consuma demasiada memoria.

1.1.4. Evaluación perezosa

No todos los lenguajes funcionales tienen los mismos mecanismos de evaluación. El lenguaje que se usará para las implementaciones (HASKELL) utiliza uno en particular: la evaluación perezosa.

La *evaluación perezosa* es una estrategia de evaluación en la que no se evalúan las expresiones cuando están vinculadas a variables, sino que su evaluación se pospone hasta que sus resultados son necesarios para otros cálculos. Por esto mismo se dice que la evaluación perezosa es *no estricta*[26].

Las tres principales características de la evaluación perezosa son:

- Los argumentos de las funciones son evaluados sólo cuando es necesario que la evaluación continúe.

  Por ejemplo, si

  ```
  f x y = x + y
  ```

  entonces

  ```
  f (6 - 2) (f 45 16)
  ⇝ (6 - 2) + (f 45 16)
  ```

  Aquí, el símbolo "⇝" indica un paso en la evaluación (o reducción) de una expresión.

  En los argumentos de la suma reemplazamos `x` por `(6 - 2)` y `y` por `(f 45 16)`. Las expresiones `(f 45 16)` y `(6 - 2)` no se evalúan antes de pasarse como argumentos.

  Para que la evaluación pueda seguir, necesitamos que se evalúen los argumentos de la suma.

  ```
  ⇝ 4 + (45 + 16)
  ⇝ 4 + 61
  ⇝ 65
  ```

- No necesariamente un argumento se evalúa completamente, sólo las partes que se necesitan son examinadas.

  Ejemplo: si

  ```
  f x y = x + 12
  ```

entonces

```
f (6 - 2) (f 45 16)
⇝ (6 - 2) + 12
⇝ 4 + 12
⇝ 18
```

En este caso `x` es sustituido por `(6 - 2)`, pero como `y` no aparece del lado derecho de la ecuación, el argumento `(f 45 16)` no aparecerá en el resultado, por lo que no se evalúa (un argumento que no se necesita no es evaluado).

Otra consecuencia de la evaluación perezosa es que es posible para el lenguaje describir **estructuras infinitas**, lo cual es parte importante de este trabajo. Hipotéticamente tales estructuras requerirían una cantidad infinita de tiempo para evaluarse o recorrerse completamente. Sin embargo, como se verá mas adelante, sólo partes finitas de estas estructuras necesitan ser examinadas. Más adelante nos concentraremos en definir listas y árboles infinitos.

1.2. Haskell

El **cálculo lambda** fue creado en la década de los 1930 por Alonso Church. Este cálculo fue inicialmente parte de un sistema más complejo que pretendía ser un fundamento matemático para la lógica. En 1935 Kleene y Rosser, estudiantes de Church, descubrieron que dicho sistema era inconsistente. Sin embargo el subsistema de términos conocido hoy como cálculo lambda fue desde entonces estudiado independientemente como un modelo de cómputo. Este cálculo se ocupa de funciones únicamente, en particular modela funciones que toman otras funciones como argumento o bien devuelven funciones como resultados. En términos de lenguajes de programación el cálculo lambda es un prototipo simple de un lenguaje funcional puro de orden superior[27].

En la década de 1950, John McCarthy define LISP (*LISt Processing*), el primer lenguaje funcional con asignación de variables, el cual tuvo muchas innovaciones que influyeron en aspectos teóricos y prácticos de la programación funcional; sin embargo LISP tuvo poca influencia del cálculo lambda. Durante la década de 1960, Peter Landin observó que un lenguaje de programación complejo puede entenderse al formularlo como un cálculo núcleo, a consecuencia de esto propone un lenguaje funcional puro llamado ISWIM (*If you See What I Mean*) que a su vez contiene al cálculo lambda como cálculo núcleo. Finalmente, ISWIM es considerado el primer lenguaje funcional puro sin asignación de variables que utiliza el cálculo lambda[24].

Gran parte del trabajo importante en lenguajes puramente funcionales se origina en el Reino Unido. En 1976 David Turner introduce SASL (*St. Andrews Static Language*), seguido en 1982 por KRC (*Kent Recursive Calculator*), en el

cual, los programas son definidos por ecuaciones recursivas; utiliza evaluación perezosa y un método de implementación basado en lógica combinatoria[24].

A mediados de la década de 1970 Robin Milner y otros desarrollan ML (*Meta Language*), que es considerado el primer lenguaje funcional moderno al incluir inferencia de tipos y polimorfismo, esto último en el sentido en que se pueden definir funciones tipadas para argumentos de diferentes tipos que siempre utilicen el mismo algoritmo. Por ejemplo, es posible definir una función que cuenta el número de elementos de una lista con elementos de cualquier tipo. Sin embargo ML usó un estilo de evaluación aplicativo como LISP, es decir, los parámetros son evaluados completamente antes de que la llamada de la función sea ejecutada[24].

Hubo otros lenguajes similares que usaron las mismas ideas pero con un punto de vista ligeramente diferente como el lenguaje HOPE, el cual usó polimorfismo tipado, donde los tipos no eran inferidos por el sistema sino que debían ser especificados por el programador. En 1986, David Turner define MIRANDA, que mezcla esas dos tendencias en un lenguaje tipado polimórficamente con evaluación perezosa y ecuaciones recursivas[24].

A mediados de la década de 1980 hubo varios investigadores habían desarrollado de forma independiente lenguajes perezosos y sus propias implementaciones para ellos. Ejemplos de lo anterior son el lenguaje ORWELL desarrollado por Philip Wadler, influenciado por KRC y MIRANDA; CLEAN, lenguaje perezoso basado explícitamente en reducción de gráficas y desarrollado por Rinus Plasmeijer. Aunque cada uno tenía ideas interesantes, no había razones para fiermar que uno era superior a cualquiera de los otros[17].

Ante tal panorama, HASKELL fue definido como un intento de crear un estándar para programación funcional, su primera versión aparece en 1990, incluye muchas características de lenguajes funcionales anteriores tales como funciones de orden superior, inferencia de tipos y evaluación perezosa. Entre sus novedades se encuentra la invención de clases de tipos (que se utilizarán en la sección 3.2). El nombre del lenguaje hace honor al lógico Haskell B. Curry, cuya obra forma la base lógica de la teoría detrás de la programación funcional[17].

Antes de describir la estructura básica de HASKELL se explicará de manera breve el concepto de **sistemas de tipos**, para entender la importancia de los lenguajes tipados. La siguiente sección se basa en un estudio publicado por Luca Cardelli sobre sistemas de tipos [5].

1.2.1. Sistemas de tipos

En lenguajes de programación, un *tipo* se define como un rango de valores que una variable (o expresión) puede tomar durante la ejecución de un programa. Los lenguajes de programación cuyas variables utilizan tipos se llaman **lenguajes tipados**.

Un *sistema de tipos* es un conjunto de reglas de inferencia de un lenguaje

tipado que realiza un seguimiento de los tipos de las variables y las expresiones en un programa, su propósito es prevenir la ocurrencia de errores de ejecución.

Un lenguaje es tipado en virtud de que exista un sistema de tipos para él. En los lenguajes explícitamente tipados los tipos son parte su sintaxis y en los implícitamente tipados no es necesario escribirlos. En lenguajes como ML o HASKELL pueden ocurrir ambas situaciones como los fragmentos de código sin información de tipos (gracias a su algoritmo de inferencia de tipos) y los fragmentos donde se especifican, ya sea voluntariamente (como forma de documentación) o en casos en los que es necesario.

Algunas ventajas de tener sistemas de tipos en un lenguaje de programación son:

- Economía en la ejecución: "*La información correcta en tiempo de compilación propicia la aplicación de operaciones apropiadas en tiempo de ejecución sin necesidad de realizar pruebas costosas*"[5].
- Desarollo en pequeña escala: "*Cuando un sistema de tipos esta bien diseñado, el chequeo de tipos puede detectar errores de programación rutinarios, lo cual evita largas sesiones de depuración*"[5].
- Economía en la compilación: "*La información de tipo se puede organizar en interfaces para módulos del programa, los módulos pueden ser compilados independientemente de los otros, cada uno depende de las interfaces de los otros. La compilación de grandes sistemas se realiza con más eficiencia*"[5].

Las principales características de los sistemas de tipos son:

- Verificables: "*Debe haber un algoritmo decidible (llamado algoritmo de verificación de tipos) que asegure que un programa se comporta bien*"[5].
- Transparentes: "*El programador puede ser capaz de predecir fácilmente cuando un programa hará la verificación correcta. Si falla en el chequeo de tipos, la razón de la falla debe ser evidente*"[5].
- Deben hacerse cumplir siempre: "*Las declaraciones de tipo deben ser revisadas estáticamente siempre que sea posible o de lo contrario revisadas dinámicamente*"[5].

En contraste con lo que ocurre en la programación orientada a objetos (cuando se realiza la conversión de tipos o *casting*), en lenguajes como HASKELL, no es posible para el programador evitar las restricciones impuestas por el sistema de tipos en la aplicación de funciones binarias (también llamadas *operadores* en HASKELL) y constructores de datos. Esa característica del lenguaje lo hace ser *fuertemente tipado*[2], en otras palabras, cada valor de un determinado tipo tiene

[2] tipado explícitamente

la garantía de haber sido creado, llamando a uno de los constructores de dicho tipo. Bajo este esquema, no serán posibles errores como por ejemplo sumar dos valores booleanos.

La ventaja de los lenguajes fuertemente tipados es que imponen un conjunto de reglas rigurosas a un programador, lo que garantiza cierta consistencia en los resultados.

Los tipos en HASKELL son inferidos automáticamente, siempre se inferirá el tipo más general de una variable. Al escribir una función de ordenamiento sin declaración de tipo, HASKELL asegura que la función trabajará para todos los valores que se puedan ordenar. En muchos casos es opcional escribir la información de tipo para mayor legibilidad del código o para escribir la documentación del programa. Sin embargo, será necesario escribirla al usar determinadas extensiones del lenguaje o al trabajar con ciertos constructores de tipo de orden superior, como se verá más adelante.

En las siguientes secciones se describirán aspectos de HASKELL en el ámbito de la programación, necesarios para entender las implementaciones de estructuras cíclicas.

1.2.2. Tipos básicos

Dado que HASKELL es puramente funcional, todos los cálculos se realizan a través de la evaluación de *expresiones* (términos sintácticos) que producen *valores* (entidades abstractas que consideramos respuestas). Todo valor tiene un tipo asociado[16].

Todos los valores de HASKELL son de primera categoría, pueden ser argumentos o resultados de funciones, o pueden ser ubicados en estructuras de datos. El proceso de asociación de un valor con su tipo se llama *tipificado*[16]. Por ejemplo:

```
 3   :: Integer
True :: Bool
'a'  :: Char
```

El símbolo "`::`" se lee "tiene el tipo".

Las principales formas de expresión de tipo son los **constructores de tipo** y las **variables de tipo**, a continuación se explicará cada una en base a la descripción consultada en [29].

Constructores de tipo

La mayoría de los constructores de tipo se escriben como un identificador que comienza con una letra mayúscula.

- `Bool, Int, Integer, Float, Double` y `Char` son (funciones constantes) constructoras de tipo (para los que se usan los mismos nombres).
- Los constructores de tipo nuevos, agregan un tipo nuevo `T` al vocabulario de tipos. Se declaran con la sentencia `data T ...` y se revisan con más detalle en la sección 1.3.6

Los constructores incorporados que tiene una sintaxis especial son los siguientes:

- *Tipos función*: `a -> b` es el tipo de las funciones del tipo `a` en el tipo `b`. Los tipos función se asocian a la derecha. Es decir:

  ```
  t1 -> t2 -> ... tn -> v
  ```

  significa

  ```
  t1 -> (t2 -> ... (tn -> v) ...)
  ```

  De manera que una aplicación de función de la forma $f\ a_1\ a_2\ \ldots\ a_n$ se asocia a la izquierda y significa $(((f\ a_1)\ a_2)\ldots)\ a_n)$

  El tipo de una función en Haskell se escribe separando mediante el constructor (`->`) los tipos de los distintos argumentos y el resultado. Por ejemplo:

  ```
  inc :: Integer -> Integer
  inc x = x + 1
  ```

  Las funciones de varios argumentos contienen varias flechas (`->`) en su declaración de tipo.

  La siguiente función toma dos enteros y devuelve el mínimo:

  ```
  minimo :: Int -> Int -> Int
  minimo x y = if x <= y then x else y
  ```

- *Tipos tupla*: `(a,b)` denota al tipo del producto cartesiano $a \times b$. Por ejemplo:

  ```
  (2,'r') :: (Int,Char)
  ```

  En particular, `()` es el tipo de 0-tuplas (correspondiente al tipo *Void* en otros lenguajes).

- *Tipos lista*: `[a]` es el tipo de listas de elementos del tipo `a`. Por ejemplo:

  ```
  [True,True,False, False] :: [Bool]
  ```

  Las listas deben ser homogéneas, es decir, todos sus argumentos deben ser del mismo tipo.

Variables de tipo

Se escriben como identificadores que comienzan con letra minúscula. Por ejemplo en el tipo `List a`, `a` es una variable de tipo que indica que el tipo de los elementos de las listas puede ser cualquiera que sustituya `a`.

1.2.3. Funciones polimórficas

Una función es **polimórfica** si su definición tiene sentido para más de un tipo, un ejemplo claro es la función identidad

```
id :: a -> a
  id x = x
```

Cualquier tipo con letras `a, b, c` (llamadas *variables de tipo* y son letras minúsculas para diferenciarlas de tipos específicos) debe entenderse como un tipo universal donde las variables están cuantificadas[3] universalmente, por ejemplo el verdadero tipo de la función identidad es

```
id :: ∀a.a → a
```

HASKELL incorpora *tipos polimórficos* en funciones. Expresiones de tipo polimórfico describen familias de tipos (ver [16]). Por ejemplo la función que obtiene la longitud de una lista puede aplicarse a diversos tipos:

```
longitud :: [Bool] -> Int
longitud :: [[Char]] -> Int
longitud :: [(Bool,Int)] -> Int
longitud :: [Float] -> Int
```

Como se puede observar sólo varia el tipo de los elementos de la lista. El tipo polimórfico que encapsula esa familia de tipos es:

```
longitud :: [a] -> Int
```

donde la función esta definida así:

```
longitud [ ] = 0
longitud (x:xs) = 1 + longitud xs
```

A continuación se presentan otros ejemplos de funciones polimórficas:

- Composición de funciones, se define de forma matemática:

[3] La cuantificación se omite en todas las definiciones.

```
(.) :: (a -> b) -> (c -> a) -> c -> b
(.) f g x = f (g x)
```

- `flip`, recibe una función de dos argumentos e invierte el orden en el que éstos se evalúan.

  ```
  flip :: (a -> b -> c) -> b -> a -> c
  flip f x y = f y x
  ```

1.2.4. Definición anónima de funciones (notación lambda)

En HASKELL las funciones también pueden definirse anónimamente a través de *abstracciones lambda* tal como en el cálculo lambda, por ejemplo la función incremento se puede definir en el cálculo lambda como $\lambda x.x + 1$ y en HASKELL como:

```
\x -> x + 1
```

(aquí la diagonal \ recuerda al símbolo λ). De manera similar, la función suma es equivalente a: l

```
\x -> \y -> x + y
```

abstracciones anidadas como ésta se pueden escribir con la notación corta

```
\x y -> x + y.
```

- Las funciones anónimas (expresiones lambda) son útiles para definir funciones que devuelven funciones como resultados, por ejemplo si se toma un elemento `x :: a` y se quiere devolver la función constante `f :: b -> a`, se pasa como parámetro el elemento fijo `x` y una variable anónima[4] `'_'`, al final se regresa `x`:

  ```
  cte :: a -> b -> a
  cte x _ = x
  ```

  Sin embargo al usar expresiones lambda se puede especificar que lo que se devuelve es una función, de esta manera la variable (anónima) que no se utiliza se pasa del lado derecho.

  ```
  cte x = \_ -> x
  ```

- También pueden usarse para dar una definición formal de una función con argumentos separados. Por ejemplo `suma x y` está realmente denotando a la función $\lambda x \to (\lambda y \to x + y)$

[4] Que denota a una variable sin nombre

- Otro uso importante es para evitar nombrar funciones a las que nos referiremos sólo una vez. Considérese el siguiente programa, el cual devuelve la lista que contiene a los primeros `n` números impares, aplicando la función `map` a la lista que va desde `0` hasta `n-1` con una función `f` que genera impares:

  ```
  impares n = map f [0..n-1] where f x = 2*x+1
  ```

  Ejemplo:

  ```
  > impares 10
      [1,3,5,7,9,11,13,15,17,19]
  ```

  Este programa puede escribirse con la función `f` explícitamente (como función anónima) delante de `map`:

  ```
  impares n = map (\x -> 2*x+1) [0..n-1]
  ```

En general, dada la variable `x` con tipo t_1 y la expresión `exp` con tipo t_2, entonces podemos decir que `\x -> exp` tiene tipo $t_1 \to t_2$.

A continuación se revisarán dos funciones importantes: *curry* y *uncurry*. Su importancia radica en que existen interpretaciones de ellas en teoría de las categorías[23] y permiten hacer *aplicación parcial*.

1.2.5. La funciones *curry*, *uncurry* y aplicación parcial

Dos funciones polimórficas de orden superior son `curry` y su contraria `uncurry`.

- La función `curry` toma una función de un único argumento el cual es un par y la convierte en una función de orden superior de dos argumentos.

  ```
  curry :: ((a, b) -> c) -> a -> b -> c
  curry f (x,y) = f x y
  ```

- La función `uncurry` toma una función binaria como la obtenida en la función anterior (curry) y devuelve una función cuyo único argumento es un par.

  ```
  uncurry :: (a -> b -> c) -> (a, b) -> c
  uncurry f x y = f (x,y)
  ```

La siguiente es una defincón de función que suma sus dos argumentos:

```
suma :: Int -> Int -> Int
suma x y  =  x + y
```

Como ocurre con la función `suma`, usualmente es mejor definir funciones con los argumentos separados (*curried functions*[5]) porque permiten hacer **aplicacion parcial**[16].

La aplicación parcial en HASKELL involucra el paso de cierto número de argumentos (menor al total) en una función de múltiples argumentos. Por ejemplo:

```
sumaDos :: Int -> Int
sumaDos = suma 2
```

En este ejemplo, la función `sumaDos` contiene el resultado de la aplicación parcial de `suma`. Es una nueva función que toma un entero y regresa el resultado de sumarle 2.

Como se mencionó en la parte de tipos básicos, el operador de tipo `->` se asocia a la derecha y la aplicación de funciones a la izquierda, por eso la firma de la función `suma` se interpreta así:

```
suma :: Int -> (Int -> Int)
```

Esto significa que `suma` toma un argumento y regresa una función que recibe otro argumento y regresa un entero. La aplicación parcial de `suma` en `sumaDos` se asocia a la izquierda y se interpreta así:

```
sumaDos 5 = (suma 2) 5
```

La aplicación parcial en funciones con argumentos separados (*curried*) da una forma efectiva de definirlas y manipular sus resultados. También es útil cuando definimos funciones de orden superior como se verá en el siguiente ejemplo:

```
dobleComp :: (a -> a -> b) -> (c -> a) -> c -> c -> b
dobleComp g f = (\x y -> g (f x) (f y))
```

La función `dobleComp` es de orden superior ya que recibe dos funciones `g` y `f`, regresa una función de dos argumentos a los que se les aplicará la función `f` y los resultados pasarán como parámetros de `g`.

Si en determinado caso conviene que `g` sea una función binaria como `suma`, creamos una nueva función (`dobleComp'`) que sea aplicación parcial de `dobleComp`:

[5] El nombre `curry` proviene de Haskell Curry, la persona que popularizó su uso.

```
dobleComp' f = dobleComp suma f
```

lo cual nos evita definir una nueva función que repita la estructura que se especificó previamente:

```
dobleComp' f = \x y -> suma (f x) (f y)
```

De igual forma podemos hacer la aplicación parcial de `dobleComp` con cualquier función binaria que necesitemos.

Existe otro mecanismo de definición de funciones en HASKELL, las **secciones**. Si se tiene un operador binario infijo `(*) :: a -> a -> a` entonces las funciones parciales obtenidas al dejar fijo uno de los argumentos de `(*)` son las *secciones* de `(*)` , su definición[29] es la siguiente[6]

```
(x*) = \y -> x * y
(*x) = \y -> y * x
```

Ejemplos de secciones son los siguientes:

- `(>2)` La función que verifica si un número es mayor que 2.
- `(3:)` La función que pone el número tres al frente de una lista.
- `(++"\n")` La función que pone una nueva línea al final de una cadena.
- `(1/)` Es la función inverso multiplicativo.

Es común utilizar secciones con la función `map` porque se evita definir funciones extras, por ejemplo:

```
> map (*3) [1,2,3,4]
   [1,6,9,12]
```

en la función anterior se evita definir otra que multiplique un entero por tres.

1.2.6. Tipos de datos nuevos

En la sección anterior se definieron los tipos básicos en HASKELL, ahora se pondrá énfasis en los tipos definidos por el usuario, llamados *tipos de datos algebraicos*.

[6] Los operadores binarios definidos con simbolos no alfabéticos se colocan entre paréntesis cuando son prefijos, por ejemplo `(*) x y` es equivalente a `x * y`. También en la declaración de tipo `(*) :: a -> a -> a`. En cambio, si el nombre del operador binario es alfabético, no se escriben los paréntesis y en los usos de notación infijo se deben usar apóstrofes izquierdos, por ejemplo `mod x y` es lo mismo que `x `mod` y`.

Para declarar tipos algebraicos se utiliza[29] la declaración `data`, la cual necesita de un nombre para el tipo T que inicie con mayúscula, parámetros $a_1, \ldots, a_n$, nombres de constructores $\texttt{cons}_1, \ldots, \texttt{cons}_\texttt{m}$ y sus parámetros b_{ik}. La forma general es:

$$\begin{aligned} \texttt{data}\ T\, a_1 \ldots a_n \ &= \ \texttt{cons}_1\ b_{11} \ldots b_{1n_1} \\ &\mid \ \texttt{cons}_2\ b_{21} \ldots b_{2n_2} \\ &\vdots \\ &\mid \ \texttt{cons}_\texttt{m}\ b_{m1} \ldots b_{mn_m} \end{aligned}$$

Esta construcción declara un constructor de tipo T y funciones constructoras $\texttt{cons}_\texttt{i}$, tales que

$$\texttt{cons}_\texttt{i} :: b_{i1} \to \ldots\ldots b_{i_{ni}} \to\ T\ a_1, \ldots, a_n$$

Existen dos casos particulares de interés, cuando hay constructores sin parámetros y cuando sólo hay un costructor con uno o varios parámetros:

- $n = n_i = 0$. En este caso los tipos se llaman enumerados, ya que se puede enumerar sus elementos y por eso son tipos finitos. La declaración sería:

  $$\texttt{data}\ T = \texttt{cons}_1 \mid \texttt{cons}_2 \mid \ldots \mid \texttt{cons}_\texttt{m}$$

  Ejemplos de esto son:

  ```
  data Carreras = Fisica | Matematicas | Ccomputacion | Actuaria

  data Estacion = Primavera | Verano | Otoño | Invierno

  data Direccion = Norte | Sur | Este | Oeste
  ```

- $m = 1$. Estos tipos se llaman tipos producto o tupla. La declaración es:

  $$\texttt{data}\ T\ a_1, \ldots, a_n = \texttt{cons}_1\ b_{11} \ldots b_{1n_1}$$

  Ejemplo:

  ```
  type Nombre = String
  type Edad = Int

  data Gente = Persona Nombre Edad
  ```

En el ejemplo anterior se define un tipo `Gente` que sólo tiene al constructor `Persona` con parámetros `Nombre` y `Edad`. Dado que el nombre de una persona es una cadena y su edad es un entero, en el ejemplo anterior se presentan dos **sinónimos de tipos** (`Nombre` y `Edad`), que son útiles para facilitar la lectura de un programa y no definen tipos nuevos, ya que sólo son nombres nuevos para ellos. Se declaran con la instrucción `type`[29].

Otro ejemplo de tipo de dato nuevo es el siguiente:

```
data Figura = Circulo Float | Rectangulo Float Float
```

El tipo figura permite definir círculos a través de su radio, o bien rectángulos los cuales deberán tener la longitud de dos de sus lados, por lo que los tipos de los constructores son:

```
Circulo :: Float -> Figura

Rectangulo :: Float -> Float -> Figura
```

1.2.7. Tipos recursivos

Como se vió en los tipos definidos por el usuario, un constructor de un tipo de dato algebraico puede tener varios campos y cada uno de éstos debe ser un tipo concreto. Teniendo eso en cuenta, podemos crear tipos cuyos campos de constructor incluyan al tipo que estamos definiendo. De esta forma, podemos crear estructuras de datos recursivas, por ejemplo:

- Los números naturales con el constructor `Cero` y el sucesor (`Suc`) que lleva como parámetro al constructor de tipo que se está definiendo (`Nat`).

  ```
  data Nat = Cero | Suc Nat
  ```

  Por ejemplo `Suc (Suc Cero)` representa al natural dos.

- El tipo `Expr` define tres constructores: `Val` con un parámetro entero; `Add` y `Mul` con dos parámetros recursivos (que definen al mismo tipo `Expr`).

  ```
  data Expr = Val Int | Add Expr Expr | Mul Expr Expr
  ```

  Esta estructura define expresiones aritméticas básicas, por ejemplo:

  ```
  Mul (Add (Val 2) (Val 7))
      (Add (Val 8) (Val 10))
  ```

  representa a la expresión $(2 + 7) \cdot (8 + 10)$

- La estructura de listas de tipo `a` es un ejemplo de tipo de dato recursivo con parámetro. Se define con los constructores `Empty` el cual representa a la lista vacía y `Cons` representa un elemento de tipo `a` seguido de una lista de tipo `List a`.

  ```
  data List a = Empty | Cons a (List a)
  ```

  Dependiendo de lo que se requiera que la lista contenga, siempre y cuando no sea de tipo `Empty`, puede producir tipos como `List Int`, `List String`, etc. Ningún valor puede tener un tipo que sea simplemente `List`, ya que éste no es un tipo por sí mismo, es un constructor de tipos. Además el constructor `Cons` llama recursivamente a `List a` que a su vez puede contener otro constructor `Cons` que llamen recursivamente a `List a` y así sucesivamente, por ejemplo:

  ```
  l1 :: List Nat
  l1 = Cons Cero (Cons (Suc Cero) (Cons (Suc (Suc Cero)) Empty))
  ```

  la cual es una lista con tres naturales de tipo `Nat`.

- `Tree a` es el constructor de tipo de una estructura de datos recursiva de árboles binarios con elementos del tipo `a` en los nodos excepto las hojas.

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

  Por ejemplo:

  ```
  t1 :: Tree Int
  t1 = Node 10 Leaf (Node 50 Leaf Leaf)
  ```

  el cual es un árbol binario de enteros con raíz igual a 10 que tiene una hoja como subárbol izquierdo. Como subárbol derecho tiene un nodo de valor 50 con hojas a los dos lados como subárboles.

A continuación se revisará el tipo de dato `Maybe`, el cual se utilizará en la definición de tipos anidados, ese tipo tiene parámetro pero no es recursivo.

1.2.8. El tipo *Maybe*

`Maybe` es un tipo de datos predefinido de HASKELL[30]. Un valor de tipo `Maybe a`, o bien contiene un valor de tipo `a` (representado como `Just a`), o es vacío (`Nothing`). Usualmente se utiliza como opción para manejar errores o casos excepcionales, sin necesidad de que el programa sea detenido.

```
data Maybe a = Nothing | Just a
```

El tipo de los constructores es:

```
 Nothing :: Maybe a
    Just :: a -> Maybe a
```

Ejemplo de uso:

```
safehead :: [a] -> Maybe a
safehead [ ] = Nothing
safehead xs = Just (head xs)
```

La función `safehead` obtiene la cabeza de una lista de tipo `a` devolviendo el constructor `Just a`, cuando la lista es vacía devuelve `Nothing` sin que se detenga la evaluación de la función.

Otro tipo de dato similar a `Maybe` es `Either`, que se utiliza para representar valores con dos posibilidades o para unir dos tipos conocidos, también puede verse como una representación de la unión ajena de conjuntos[1]. Un valor de tipo `Either a b`, o bien es `Left a` o es `Right b`.

```
data Either a b = Left a | Right b
```

El tipo de los constructores es:

```
   Left :: a -> Either a b
  Right :: b -> Either a b
```

Por ejemplo podemos definir listas heterogéneas usando `Either`

```
[Left 'x', Right True, Left 'y'] :: [Either Char Bool]
```

O como un tipo que maneja un caso de excepción:

```
emparejar :: Int -> Either Int String
emparejar gente
   | gente < 0 = Right "Número negativo de personas."
   | gente > 30 = Right "Demasiadas personas."
   | even gente = Left (gente `div` 2)
   | otherwise = Right "Número impar de personas."
```

[1]Para dos conjuntos A y B se define la *unión ajena* de A y B como $A \sqcup B = (A \times \{x\}) \cup (B \times \{y\})$, donde $x \notin B$ y $y \notin A$.

En este ejemplo `emparejar` indica cuantos grupos se obtendrán si se empareja a cierto número de personas en alguna actividad. También permite saber si se tienen demasiados participantes para la actividad (en este caso más de treinta) o si alguien se queda sin grupo. Entonces `emparejar` regresará un entero que representa el número de grupos que se tienen (`Left Int`), o bien una cadena que describe la razón por la cual no se pueden crear los grupos (`Right String`). La función que aplica `emparejar` es la siguiente:

```
grupos :: Int -> String
grupos gente = case emparejar gente of
         Left grupos -> "Tenemos " ++ show grupos ++ " grupo(s)."
         Right problema -> "Problema! " ++ problema
```

En la función `grupos`, la expresión `case` distingue entre varias alternativas de un valor (`Left` o `Right` en este caso) y devuelve el resultado adecuado con el uso de ajuste de patrones (ver sección 1.1.1), como se oberva en los siguientes ejemplos de evaluación:

```
> grupos 13
   "Problema! No puede emparejarse un número impar de personas."

> grupos 28
   "Tenemos 14 grupo(s)."
```

Las guardias son usadas para dar alternativas en la definición de funciones de acuerdo a expresiones booleanas arbitrarias. Las guardias aparecen después de todos los parámetros de una función pero antes del signo igual, y comienzan con una barra vertical. La forma general de funciones definidas con guardias es la siguiente[26] [7]:

```
    fun x1 x2 ... xk
       | g1        = e1
       | g2        = e2
         ...
       | otherwise = e
```

La expresión `let` también es muy común, se utiliza para realizar definiciones locales en una expresión. Por ejemplo:

```
let x = 3 + 2
       in x^2 + 2*x - 4

⇝ (3+2)^2 + 2*(3+2) - 4
```

[7] El uso de `otherwise` no es obligatorio.

Al igual que `Maybe` y `Either`, existen diversos tipos predefinidos de HASKELL que incluyen algunas funciones útiles, por ejemplo para el tipo `List` se tiene la función `permutations` que devuelve una lista con todas las permutaciones de la misma o `reverse` que regresa la misma lista con sus elementos en orden inverso. Para poder ser utilizados deben ser importados en nuestros programas.

A continuación se presentan los operadores de plegado y su utilidad en la programación funcional. Gran parte de la siguiente sección se basa en el artículo de Graham Hutton *"A Tutorial on the Universality and Expressiveness of Fold"*[18].

1.3. Operadores de plegado

Muchos programas que involucran repetición se expresan naturalmente usando alguna forma de recursividad, y sus propiedades se verifican usando alguna forma de inducción. En programación funcional, la recursión y la inducción son las principales herramientas para definir y probar propiedades de programas.

No es de sorprenderse que los programas recursivos compartan un patrón común de recursión, y sus pruebas inductivas un patrón de inducción. Tal repetición puede evitarse mediante la introducción de operadores especiales de recursión (de orden superior) que encapsulen patrones comunes, lo que permite que nos concentremos en las partes que son diferentes en cada aplicación.

En programación funcional, *foldr* es un operador de recursión que encapsula un patrón de recursión para procesamiento de listas. Este operador tiene su origen en la teoría de la recursión (Kleene, 1952), mientras que su uso como concepto central en programación funcional se remonta al operador de reducción en el lenguaje APL (*Array Processing Language*) (Iverson, 1962) [18].

El patrón de recursión de *foldr* mencionado anteriormente es el siguiente: si se pasa como tercer argumento de la función a la lista vacía, se devuelve cierto valor base dado como segundo argumento (correspondiente al caso base de la definición). De lo contrario, cierta función binaria (dada como primer argumento) se aplica a la cabeza, como primer argumento, y al valor resultante de aplicar la llamada recursiva a la cola.

En HASKELL el operador `foldr` se define como sigue:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [ ]     = v
foldr f v (x:xs)  = f x (foldr f v xs)
```

De manera que cualquier función que cumpla el esquema anterior puede definirse simplemente como:

```
fun = foldr g e
```

donde `g` es el operador que se ha de aplicar y `e` el valor base[8].

Por ejemplo, la función que suma una lista de naturales usualmente se define de esta forma:

```
sumList :: [Nat] -> Nat
sumList [ ]    = 0
sumList (x:xs) = x + (sumList xs)
```

Se simplifica utilizando de `foldr` de esta forma:

```
sumList = foldr (+) 0
```

Un ejemplo de evaluación:

foldr (+) 0 [1,2,3,4]
$\rightsquigarrow$ 1 + (*foldr* (+) 0 [2,3,4])
$\rightsquigarrow$ 1 + (2 + (*foldr* (+) 0 [3,4]))
$\rightsquigarrow$ 1 + (2 + (3 + (*foldr* (+) 0 [4])))
$\rightsquigarrow$ 1 + (2 + (3 + (4 + (*foldr* (+) 0 [ ]))))
$\rightsquigarrow$ 1 + (2 + (3 + (4 + 0)))
$\rightsquigarrow$ 10

Otros ejemplos del uso de este operador son los siguientes:

- Producto de los elementos de una lista

  ```
  product :: [Int] -> Int
  product = foldr (*) 1
  ```

- Operador *and* a los elementos de una lista

  ```
  and :: [Bool] -> Bool
  and = foldr (&&) True
  ```

- Juntar los elementos de una lista de listas en una sola lista

  ```
  concat :: [[a]] -> [a]
  concat = foldr (++) [ ]
  ```

- Reversa de una lista

  ```
  reverse :: [a] -> [a]
  reverse = foldr (\x xs -> xs ++ [x]) [ ]
  ```

- *map* de una lista

  ```
  map :: (a -> b) -> [a] -> [b]
  map f = foldr (\x xs -> f x : xs) [ ]
  ```

[8] Es importante mencionar que `foldr` es una función de orden superior ya que uno de sus parámetros en una función.

- Filtrar los elementos de una lista con cierto criterio

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x xs -> if p x then x:xs else xs) [ ]
```

Las principales ventajas de utilizar operadores de plegado son:

- El programador sólo tiene que proporcionar el valor base y la función binaria para el problema correcto.
- No se necesita poner atención a la recursividad ya que ésta se encuentra encapsulada en el operador de plegado.

A continuación se presenta la propiedad universal de *foldr*, con la que se pueden mostrar propiedades de manera similar a la manipulación algrabraica de ecuaciones.

1.3.1. Propiedad universal

Con listas finitas, la **propiedad universal de foldr** se enuncia como "*una equivalencia entre dos definiciones para una función g que procesa listas*"[18]:

Para cualquier $e :: b$, cualquier operador $f :: a \to b \to b$ y cualquier función estricta[2] $g :: [a] \to b$ se tiene la siguiente equivalencia:

$\forall x :: a, \forall xs :: [a]$

$$\begin{array}{lll} g & [\,] & = v \\ g & (x : xs) & = f\ x\ (g\ xs) \end{array} \quad \text{si y sólo si} \quad g = foldr\ f\ v$$

La justificación de la equivalencia se expone a continuación, para la implicación de derecha a izquierda, se observa que al sustituir $g = foldr\ f\ v$ en las dos ecuaciones de g nos da como resultado la definición recursiva de $foldr$, en la implicación de derecha a izquierda, las dos ecuaciones de g son precisamente las suposiciones requeridas para mostrar que $g = foldr\ f\ v$ usando inducción para listas finitas. Ambos razonamientos en conjunto implican que $foldr\ f\ e$ es solución única (respecto a la incógnita g), de las ecuaciones a la izquierda de la propiedad universal.

De esta manera la propiedad universal de *foldr* encapsula un patrón de pruebas referente a listas finitas, al igual que el operador *foldr* en sí mismo encapsula un patrón de recursión para el procesamiento de listas.

El uso principal de esta propiedad es evitar la realización de pruebas inductivas como se observa en el siguiente ejemplo:

[2]Una función es estricta si $f(\perp) = \perp$, donde $\perp$ o *bottom* representa una expresión que no regresa un valor normal (cae en un ciclo infinito o devuelve un error de ejecución). Una función de varios argumentos puede ser estricta en cada argumento o en todos.

Para demostrar que:

$$(+1) \cdot sumList = foldr\ (+)\ 1$$

donde *sumList* es la función que suma los elementos de una lista, sustituimos $g = (+1) \cdot sumList$, $f = (+)$ y $v = 1$. Acudiendo a la propiedad universal, se concluye que la ecuación por probar es equivalente a las siguientes dos ecuaciones:

$$\begin{array}{lll} ((+1) \cdot sumList) & [\,] & = 1 \\ ((+1) \cdot sumList) & (x : xs) & = x + ((+1) \cdot sumList)\ xs \end{array}$$

Que es lo mismo que:

$$\begin{array}{lll} sumList & [\,] & + 1 = 1 \\ sumList & (x : xs) & + 1 = x + (sumList\ xs + 1) \end{array}$$

Y así, se pueden verificar las igualdades:

$$\begin{array}{ll} & sumList\ [\,] + 1 \\ = & \quad \langle \text{ Definición de } sumList \rangle \\ & 0 + 1 \\ = & \quad \langle \text{ Aritmética} \rangle \\ & 1 \end{array} \qquad \begin{array}{ll} & sumList\ (x : xs) + 1 \\ = & \quad \langle \text{ Definición de } sumList \rangle \\ & (x + sumList\ xs) + 1 \\ = & \quad \langle \text{ Aritmética} \rangle \\ & x + (sumList\ xs + 1) \end{array}$$

Además de ser usada como principio de prueba, la propiedad universal de *foldr* se puede utilizar como *principio de definición*, que nos guía en la transformación de funciones recursivas en definiciones que involucran *foldr*. Por ejemplo, consideremos la función recursiva *map f* que aplica la función f a cada elemento de la lista:

$$\begin{array}{lll} map & :: & (a \to b) \to [a] \to [b] \\ map\ \ f\ \ [\,] & = & [\,] \\ map\ \ f\ \ (x : xs) & = & f\ x\ :\ map\ f\ xs \end{array}$$

Para redefinir *map f* con *foldr*, se debe resolver la ecuación $map\ f = foldr\ g\ v$ para una función g y un valor v. Usando la propiedad universal, concluimos que esa ecuación es equivalente a las siguientes dos ecuaciones:

$$\begin{array}{lll} map\ \ f\ \ [\,] & = & v \\ map\ \ f\ \ (x : xs) & = & g\ x\ (map\ f\ xs) \end{array}$$

En la primera ecuación es claro que $v = [\,]$ por definición de *map*. En la segunda, calculamos una definición de g del siguiente modo:

$$\begin{aligned}
& map\ f\ (x : xs) = g\ x\ (map\ f\ xs) \\
= & \quad \langle \text{ Definición de } map \rangle \\
& f\ x : map\ f\ xs\ =\ g\ x\ (map\ f\ xs) \\
= & \quad \langle \text{ Generalizando } (map\ f\ xs) \text{ a } ys \rangle \\
& f\ x : ys\ =\ g\ x\ ys \\
= & \quad \langle \text{ Abstracción lambda } \rangle \\
& g\ =\ \backslash x\ ys \rightarrow f\ x : ys
\end{aligned}$$

Por lo anterior, usando la propiedad universal hemos calculado que:

$$map\ f\ =\ foldr\ (\backslash x\ ys \rightarrow f\ x : ys)\ [\,]$$

Al utilizar la propiedad universal como principio de definición, se simplifica el proceso de establecer programas, también reducimos su tamaño y facilitamos su depuración.

Teorema de fusión

Como consecuencia de la propiedad universal se tiene el **teorema de fusión** del operador `fold`:

$$\begin{array}{ll} h\ w & = v \\ h\ (g\ x\ y) & = f\ x\ (h\ y) \end{array} \qquad \text{si y sólo si} \qquad h \cdot foldr\ g\ w\ =\ foldr\ f\ v$$

Llamado así porque se fusiona el plegado $foldr\ g\ w$ seguido de la función h en un único plegado $foldr\ f\ v$.

"Este modelo de ecuación ocurre frecuentemente cuando razonamos con programas[9] *que contienen a foldr"*[18]. Como ejemplo probaremos la ecuación que afirma que el operador *map* se distribuye sobre la composición de funciones:

$$map\ f\ \cdot\ map\ g\ =\ map\ (f \cdot g)$$

Reemplazando la segunda y tercera presencias de *map* en la ecuación por su definición usando *foldr* (dada anteriormente), la ecuación se reescribe en una forma que encaje con la conclusión del teorema de fusión:

$$map\ f\ \cdot\ foldr\ (\lambda x\ xs \rightarrow g\ x\ :\ xs)\ [\,]$$
$$=$$
$$foldr\ (\lambda x\ xs \rightarrow (f \cdot g)\ x : xs)\ [\,]$$

[9] *"Reasoning about programs"* se refiere a verificar los programas en base a determinado sistema formal, utilizando herramientas de ayuda para demostrar teoremas[14, 19].

Aplicando el Teorema de Fusión y simplificando llegamos a las siguientes dos ecuaciones, que son ciertas por definición de *map* y $(\cdot)$

$$\begin{array}{lllll} map & f & [\,] & = & [\,] \\ map & f & (g\ x : y) & = & (f \cdot g)\ x : map\ f\ y \end{array}$$

Como ocurre con el principio universal, la aplicación principal de la propiedad de fusión es al utilizarla como principio de prueba, y con ello evitar las pruebas inductivas.

De la misma forma en que ocurre con la función `curry`, el operador `foldr` tiene su contrario (`unfoldr`), el cual es relevante en este trabajo ya que más adelante será de mucha utilidad para desplegar estructuras cíclicas.

1.3.2. Unfoldr

El operador *unfoldr* toma un valor inicial de tipo `b` y le aplica una función de tipo `b -> Maybe(a,b)` para generar una lista de tipo `a`. La biblioteca estándar de Haskell para listas define el operador `unfoldr`[30] como:

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f u = case f u of
     Nothing -> [ ]
     Just (x,v) -> x : (unfoldr f v)
```

Mientras `foldr` reduce una lista a un valor, `unfoldr` crea una lista a partir de un valor de inicio `u`, mediante una función de opción `b -> Maybe (a,b)` en donde el caso `Nothing` indica cuándo comienza la construcción de la lista (a partir de la lista vacía y las aplicaciones pendientes de `(:)`). En el caso `Just (x,v)` se pone `x` al inicio de la lista y recursivamente aplicamos `unfoldr` a la misma función de opción y al valor de `v` modificado. Por ejemplo:

```
unfoldr (\b -> if b == 0
               then Nothing
               else Just (2*b, b-1)) 10
```

devuelve la lista `[20,18,16,14,12,10,8,6,4,2]`, el valor de inicio es `10` y se genera una lista con los números menores que diez multiplicado cada uno por dos.

El operador `unfoldr` también puede definir estructuras potencial o estrictamente infinitas, un ejemplo de lista estrictamente infinita es la generada por la función `iterate`[30], la cual se define normalmente así:

```
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : iterate f (f x)
```

de modo que

```
iterate f x = [x, f x, f (f x), f (f (f x)), ...
```

Regresa una lista infinita de aplicaciones repetidas de `f` a `x`. Con `unfoldr` se define de la siguiente forma:

```
iterate f = unfoldr (\x -> Just (x, f x))
```

Como la lista nunca termina, no es necesario definir una condición para que se detenga (es decir, un caso para `Nothing`).

En algunos casos es conveniente escribir la función de opción como tres funciones, una que indique la condición para detenerse y las otras dos para producir los componentes del par:

```
unfoldr :: (b -> Bool) -> (b -> a) -> (b -> b) -> b -> [a]
unfoldr p f g x
    | p x       =  [ ]
    | otherwise  =  f x : (unfoldr  p f g (g x))
```

De este modo se evita el uso del tipo `Maybe` al hacer explícitas las condiciones con que se genera la lista. Por ejemplo:

```
cuad = unfoldr (\x -> x>10) (\x -> x*x) (\x -> x+1)
```

La función `cuad` genera la lista de los cuadrados de ciertos números, desde el que definamos como inicial, hasta el diez. La primera función `\x -> x>10` comprueba la condición de ser menor que diez, despues `\x -> x*x` eleva al cuadrado el número actual y por último `\x -> x+1` incrementa el valor para la siguiente llamada recursiva, por ejemplo:

```
> cuad 1
     [1,4,9,16,25,36,49,64,81,100]

> cuad 4
     [16,25,36,49,64,81,100]
```

De manera similar a lo que sucede con el principio universal de *foldr*, el principio universal de *unfoldr* encapsula un patrón de pruebas referente a listas.

Propiedad universal de unfoldr

Para cualesquiera funciones $p :: b \to Bool$, $f_1 :: b \to a$ y $f_2 :: b \to b$ y una función estricta cualquiera $g :: b \to [a]$ se tiene la siguiente equivalencia:

$\forall x :: b$

$$\begin{array}{lll} g\ x = if\quad p\ x & & \\ \quad then\quad [\,] & \text{si y sólo si} & g\ =\ unfoldr\ p\ f_1\ f_2 \\ \quad else\ (f_1\ x) : g\ (f_2\ x) & & \end{array}$$

La propiedad universal de unfold implica que $g\ =\ unfoldr\ p\ f_1\ f_2$ es solución única (respecto a la incógnita g), de la ecuación a la izquierda de la propiedad universal.

El operador unfoldr en otras estructuras

Es posible definir *unfoldr* con otras estructuras de datos, por ejemplo con `Tree`, que es un tipo genérico de árboles binarios con datos en los nodos y en las hojas:

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

el operador unfoldr para `Tree` utiliza el tipo opción `Either` (predefinido en Haskell) porque necesitamos un parámetro extra en el caso básico (para listas es la lista vacía pero para árboles es el tipo de la hoja):

```
unfoldT :: (b -> Either a (b,a,b)) -> b -> Tree a
unfoldT f u = case f u of
                  Left a -> Leaf a
                  Right (l,x,r)
                        -> Branch (unfoldT f l) x (unfoldT f r)
```

Por ejemplo, si queremos generar el árbol de 3 niveles que tenga ceros en las hojas y unos en los demás nodos lo hacemos de la siguiente forma:

```
unfoldT (\b -> if b == 0
               then Left b
               else Right(b-1,1,b-1)) 3
```

Capítulo 2

Estructuras de datos puramente funcionales

En el capítulo anterior se caracterizó a la programación funcional y en particular al lenguaje HASKELL con algunos detalles de sintaxis y de tipos. En este capítulo se presentan estructuras de datos funcionales infinitas y cíclicas.

2.1. Estructuras funcionales vs imperativas

Históricamente, el desarrollo de los lenguajes funcionales ha sido más lento que el de los imperativos, esto se debe en parte al costo que conlleva hacer un análisis matemático para preservar propiedades como la pureza en un programa funcional o realizar la verificación del mismo. Se puede encontrar mucha documentación sobre estructuras de datos imperativas en lenguajes como C, C++ o JAVA, algunas de ellas ampliamente optimizadas, no es así en el caso funcional.

Entonces nos preguntamos ¿por qué diseñar estructuras puramente funcionales es complicado? primero porque desde el punto de vista del diseño e implementación de estructuras de datos funcionales, el no poder usar asignaciones (y por tanto, no poder hacer actualizaciones destructivas) es una gran desventaja. Los cambios destructivos pueden ser peligrosos cuando se utilizan indebidamente, pero son muy efectivos cuando se utilizan adecuadamente. Las estructuras de datos imperativas a menudo se basan en asignaciones, por lo que debemos usar soluciones distintas para estructuras funcionales. La segunda dificultad es que se espera que las estructuras de datos funcionales sean más flexibles que sus contrapartes imperativas. En particular, cuando actualizamos una estructura de dato imperativa, solemos aceptar que la versión anterior de la estructura ya no estará disponible, pero cuando actualizamos una estructura funcional, esperamos que ambas, la anterior y la nueva versión, estén disponibles para su posterior procesamiento[22].

Una estructura de datos que admite múltiples versiones es llamada *persistente* mientras que una estructura que sólo permite una única versión es llamada *efímera*[6]. Los lenguajes de programación funcional tienen la propiedad de que todas las estructuras de datos son persistentes de forma automática.

La *persistencia parcial* permite realizar modificaciones sólo para la estructura de datos actual, pero permite las consultas a cualquier versión anterior. La *persistencia completa* permite realizar consultas y modificaciones a todas las versiones anteriores de la estructura de datos. Con este tipo de persistencia las versiones no forman una trayectoria lineal simple, forman un árbol de versiones[6].

"Las estructuras de datos persistentes ofrecen ventajas que no comparten sus contrapartes efímeras"[4]:

- *"Ir hacia atrás y ver cualquier versión previa de la estructura de datos.*
- *Ir hacia atrás y hacer una secuencia de modificaciones a una versión previa de la estructura de datos, sin afectar a la versión actual (esto permite un árbol de versiones).*
- *Combinar estructuras persistentes, como tomar la unión de dos conjuntos persistentes.*
- *Paralelismo implícito, es decir, la habilidad de ver y modificar en paralelo diferentes versiones de una estructura de datos."*[4]

La manera de proporcionar la persistencia es hacer una copia de la estructura de datos cada vez que se cambia, esto tiene el inconveniente de que requiere un espacio y un tiempo proporcional al espacio ocupado por la estructura de datos original.

Las estructuras de datos imperativas son típicamente efímeras, pero cuando se requiere una estructura persistente, ésta será en general más complicada y tal vez incluso (asintóticamente) menos eficiente que su equivalente efímera. Los teóricos han establecido cotas mínimas que sugieren que los lenguajes funcionales son menos eficientes que los imperativos en ciertas situaciones. Sin embargo es posible a menudo diseñar estructuras de datos funcionales que son (asintóticamente) tan eficientes como las soluciones imperativas[22].

2.2. Listas infinitas

Una consecuencia importante de la evaluación perezosa es la posibilidad de definir estructuras infinitas. Intuitivamente, éstas requieren una cantidad infinita de tiempo para recorrerse al ser evaluadas, sin embargo bajo la evaluación perezosa es posible hacer cómputos sólamente con porciones de ellas, en lugar de objetos completos. A continuación veremos ejemplos de listas, ya que éstas son las estructuras infinitas más usadas.

Un ejemplo simple de lista infinita es la lista constante:

```
listunos = 1 : listunos
```

cuya evaluación es una lista potencialmente infinita:

```
[1,1,1,1,1,1,1,1,1,1,1,...
```

Se aplicará una función a la lista anterior:

```
unoydos : : [Int] -> Int
unoydos (x:y:zs) = x + y
```

La función `unoydos` suma los primeros dos elementos de la lista. Se aplica sobre una lista infinita sin necesidad de recorrerla en su totalidad.

```
unoydos listunos
⇝ unoydos (1:listunos)
⇝ unoydos (1:1:listunos)
⇝ 1 + 1
⇝ 2
```

Ejemplos de listas similares son las funciones `from` y `fromStep`:

- ```
  from :: Int -> [Int]
  from n = n : from (n + l)
  ```

  Esta función recibe un entero `n` y regresa la lista de los enteros a partir de `n`, por ejemplo:

  ```
  from 5
  ⇝ 5 : from 6
  ⇝ 5 : 6 : from 7
  ⇝ ...
  ```

  nos da la lista `[5,6,7,8,9,10,11...`

- ```
  fromStep :: Int -> Int -> [Int]
  fromStep n m = n : fromStep (n + m) m
  ```

  Esta función recibe dos enteros `n`,`m` y regresa una lista enteros a partir de `n` con saltos de `n-m`, por ejemplo:

  ```
  fromStep 3 2
  ⇝ 3 : fromStep 5 2
  ⇝ 3 : 5 : fromStep 7 2
  ⇝ ...
  ```

  nos da la lista `[3,5,7,9,...`

Es importante destacar que las funciones `unoydos`, `from` y `fromStep` no funcionarían sin el mecanismo de evaluación perezosa. Ya que la función `(:)`

agrega elementos a la lista conforme los va necesitando. Si no hubiera evaluación perezosa, el programa entraría en un ciclo infinito antes de devolver la lista resultado.

Las funciones `from` y `fromStep` estan predefinidas en HASKELL bajo los nombres `enumFromThen` y `enumFromThenTo`[30] que a su vez estan incluidas como azúcar sintáctica[1] en la sintaxis de listas `[ ]`:

- `enumFromThen` la cual puede escribirse como `[n,m..]`, ésta produce una lista (potencialmente infinita) de valores. Las dos primeras posiciones son `n` y `m`, los siguientes elementos de la lista vienen marcados por la diferencia `m - n`. Por ejemplo:

  ```
  [1,3..] = [1, 3, 5, 7,...]
  ```

- `enumFromThenTo` la cual puede escribirse como `[n,m..p]`, ésta produce una lista (potencialmente infinita) de valores. Las dos primeras posiciones son `n` y `m`, los siguientes elementos de la lista vienen marcados por la diferencia `m - n`, la lista tiene valor límite `p`, por lo cual es finita. Por ejemplo:

  ```
  [1,3..20] = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
  ```

Si se prueba que cada elemento de una lista infinita puede calcularse en un tiempo finito (es decir, que el procedimiento que lo genera termina) se dice que la lista es **productiva** (*list productivity*, ver [25]).

Una versión infinita de un programa puede ser más abstracta y más simple de escribir, ya que no es necesario saber de antemano la longitud de la lista. Consideremos el problema de encontrar el n-ésimo número primo usando la criba de Eratóstenes[2]. Si trabajamos con listas finitas, debemos saber qué tan larga es la lista que se requiere para acomodar los primeros n primos, si trabajamos con una lista infinita, eso no es necesario[3].

A continuación se presenta la versión usando la lista finita:

```
primosHasta m = 2 : eratos [3,5..m]  where
   eratos [ ]     = [ ]
   eratos (p:xs) = p : eratos (xs `minus` [p, p+2*p..m])
```

[1] *syntactic sugar* es un término acuñado por Peter Landin para describir las adiciones de sintaxis a un lenguaje, que dan al programador una forma alternativa de escribir código en una notación familiar o más concisa.

[2] La criba de Eratóstenes es un procedimiento que permite hallar todos los números primos menores que un número natural dado. Se parte de una lista de números que van desde el dos hasta un determinado número, se elimina de la lista los múltiplos de dos, luego tomamos el primer número después del dos que no fué eliminado (el tres) y se elimina de la lista sus múltiplos, y así sucesivamente. Los números que permanecen en la lista son los primos.

[3] Porque sólo la parte necesaria de la lista será generada como producto de un cómputo.

La idea de esta función es la siguiente se agrega el número `2` y se procesa la lista de impares desde tres hasta `m` (`[3,5..m]`) con la función recursiva `eratos`, la cual revisa si se alcanzó el final de la lista en el caso base o agrega el primer elemento (`p`) de aquella lista de impares seguido de la llamada recursiva sobre la misma lista sin los múltiplos impares de `p`. Para quitar estos múltiplos utilizamos la función `minus`. De este modo la función `eratos` agrega `3` a la lista y descarta los multiplos impares de tres, después agrega `5` y descarta los múltiplos impares de cinco, y así sucesivamente, como en el procedimiento realizado en la criba.

La función `minus` quita de una lista creciente los elementos que tiene en común con otra lista creciente del mismo modo en que se hace la diferencia de conjuntos.

```
minus (x:xs) (y:ys) | x < y  = x : minus xs (y:ys)
                    | x == y = minus xs  ys
                    | x > y  = minus (x:xs) ys

minus xs   _   = xs
```

el último renglón de `minus` regresa la lista actual y detiene la recursión cuando alguna de las listas es vacía, lo cual ocurre al trabajar con listas finitas. Esa condición se puede omitir si trabajamos con una listas infinitas, como sucede en la siguiente versión de la función (donde se quita el máximo de la lista).

```
primos = 2 : eratos [3,5..]  where
   eratos [ ]     = [ ]
   eratos (p:xs) = p : eratos (xs `minus` [p, p+2*p..])
```

esta versión muestra el potencial de las listas infinitas en la solución de problemas de esta índole.

Números de Fibonacci

Para generar la lista infinita de la sucesión de Fibonacci[4] en Haskell, definimos la función `fibs`:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

La función `zipWith (+)` suma los elementos de dos listas uno a uno y los junta en una nueva, por ejemplo:

`zipWith (+) [1,2,3] [4,5,6]` $\rightarrow$ `[5,7,9]`

[4]La *sucesión de Fibonacci:* $1, 1, 2, 3, 5, 8$, etc., donde el n-ésimo término de la sucesión se obtiene sumando los dos términos previos y los dos primeros términos son 1.

Para generar la sucesion de Fibonacci basta con colocar los primeros dos números de la sucesión (1 y 1) al principio de la lista, y después trabajar con ésta y su cola, sumando los valores de las dos listas uno a uno con la función `zipWith (+)`:

`fibs` $\rightarrow [1\ ,\ 1\ ,\ 2\ ,\ 3\ ,\ 5\ ,\ \ 8\ ,\ 13\ ,\ 21\ ,\ 34\ ,\ 55\ ,\ \ 89\ , ...$

`tail fibs` $\rightarrow [1\ ,\ 2\ ,\ 3\ ,\ 5\ ,\ 8\ ,\ 13\ ,\ 21\ ,\ 34\ ,\ 55\ ,\ 89\ ,\ 144\ , ...$

`zipwith (+) fibs (tail fibs)`

$\rightarrow [1+1=\mathbf{2}, 1+2=\mathbf{3}, 2+3=\mathbf{5}, 3+5=\mathbf{8}, 5+8=\mathbf{13}, ...$

`1 : 1 : zipwith (+) fibs (tail fibs)`

$\rightarrow [\ \mathbf{1}\ ,\ \mathbf{1}\ , 1+1=\mathbf{2}, 1+2=\mathbf{3}, 2+3=\mathbf{5}, 3+5=\mathbf{8}, 5+8=\mathbf{13}, ...$

Lo anterior es posible gracias al mecanismo de evaluación perezosa, ya que la lista `fibs` puede ser autorreferenciada al pedir únicamente los dos primeros valores, que son de hecho los argumentos de la función `zipWith (+)` y no toda la lista.

Números de Hamming

La secuencia de los números de Hamming consiste en todos los números de la forma

$$2^a \cdot 3^b \cdot 5^c$$

para enteros no negativos a, b y c, en otras palabras, números cuyos únicos factores primos son 2, 3 y 5. Los primeros elementos de la secuencia son los siguientes:

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, 40, 45, ...$$

Para generar la lista infinita de los números de Hamming, se agrega el número 1 a la lista y se trabaja simultáneamente con la misma lista en tres instancias. Una que la multiplica por 2, otra por 3 y la última por 5, de modo que cualquier combinación de productos de esos números es revisada. Por último esas tres instancias se unen y los números repetidos aparecen una sola vez en el resultado

```
hamming = 1 : map (2*) hamming
      `union` map (3*) hamming
      `union` map (5*) hamming
```

La función `union` une dos listas crecientes de enteros y sin repetir elementos, como en la operación de unión de conjuntos.

```
union (x:xs) (y:ys) | x < y = x : union xs (y:ys)
| x == y = x : union  xs  ys
| x > y = y : union  (x:xs)  ys
```

si se requiere darle uso a `union` con listas finitas, basta con agregar una condición similar a la de la función `minus` (auxiliar para generar la lista de primos):

```
union  xs  ys  =  xs ++ ys
```

que maneja el caso cuando las dos listas son finitas.

El hecho de quitar las condiciones para detener la recursión en estructuras de datos (por medio de los constructores) o en las funciones definidas sobre ellas (mediante ajuste de patrones, como ocurre con la función `union`) es una de las características principales de las estructuras infinitas.

Ahora se profundizará en un tipo especial de estructura infinita, la estructura cíclica, la cual es muy importante en el desarrollo sucesivo de este trabajo ya que se presentarán definiciones de ésta con tipos anidados.

2.3. Estructuras cíclicas

Cuando una o varias partes de una estructura infinita se repite con regularidad decimos que es una estructura cíclica. Un ciclo es una subestructura que se repite a si misma a través de un apuntador.

Primero limitaremos nuestra atención a las listas cíclicas. Las listas son estructuras lineales, así una lista infinita puede ser o no ser cíclica.

Por ejemplo, la listas

`[1,2,3,4,5,...`

`[3,5,7,5,5,7,5,5,5,7,5,5,5,5,7,...`

no contienen ciclos, mientras que las listas

`[1,2,1,2,1,2,...`

`[1,2,3,2,3,2,3,...`

en donde ambas listas son cíclicas.

Cuando se menciona un apuntador en una estructura cíclica, se habla de la idea de una flecha que va de un punto a otro de la estructura de forma que se genera un ciclo, no se debe confundir con la idea bien conocida e intuitiva de programación imperativa, en la que *apuntador* es una variable que da referencia a una región en memoria. Si bien el concepto de apuntador como variable en programación imperativa puede resolver el problema de un ciclo en una estructura de datos, en programación funcional debemos seguir otro camino ya que no se tiene asignación de variables.

Empezaremos aproximando la solución del problema de la representación de apuntadores en Haskell con herramientas propias de la programación funcional, como se verá a continuación.

Para generar ciclos en una lista nos podemos ayudar del operador fix:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Este operador se definió en el lenguaje PCF (*Programming Computable Functions*) como un operador primitivo de punto fijo que permite la autoaplicación, y de esta manera se pueden definir funciones recursivas con nombre (ver [27]).

Su uso para definir listas cíclicas lo vemos a continuación:

- `clist1 = fix (\xs -> 1 : 2 : xs)`

  representa a la lista `[1,2,1,2,1,2,...`
- `clist2 = 1 : fix (\xs -> 2 : 3 : xs)`

  representa la lista `[1,2,3,2,3,2,3,2,3,...`

La definición de fix es en sí misma equivalente a la definición de un punto fijo ($F(x) = x$). Para entender mejor esa idea, a continuación se presenta un fragmento de la evaluación de `clist1`:

```
clist1 = fix (\xs ->1 : 2 : xs)
⇝ (\xs -> 1 : 2 : xs) (fix (\xs -> 1 : 2 : xs))
⇝ 1 : 2 : fix ((\xs -> 1 : 2 : xs)
⇝ 1 : 2 : (\xs -> 1 : 2 : xs) (fix (\xs -> 1 : 2 : xs))
⇝ 1 : 2 : 1 : 2 : fix ((\xs -> 1 : 2 : xs)
⇝ 1 : 2 : 1 : 2 : (\xs -> 1 : 2 : xs) (fix (\xs -> 1 : 2 : xs))
⇝ 1 : 2 : 1 : 2 : 1 : 2 : fix ((\xs -> 1 : 2 : xs)
```

⋮

Pero esta representación no ofrece medios explícitos para manipular la estructura, es decir, al no tener un constructor de tipo sobre estas listas sólo se pueden utilizar funciones de listas predefinidas de Haskell y no podemos definir un operador de plegado (*unfoldr*).

En el artículo *"Revisiting catamorphisms over datatypes with embedded functions"* escrito por Fegaras y Sheard [7], se propone una representación de estructuras cíclicas como estructuras de datos de orden superior (que tienen funciones en los constructores de tipo). La idea es representar ciclos por medio de expresiones explícitas de punto fijo en la declaración de tipo. De este modo, un tipo de dato cíclico `C` tiene un constructor extra `Rec :: (C -> C) -> C`. Para listas cíclicas tenemos la siguiente definición que representa listas finitas o listas cíclicas.

```
data Clist  = Nil
            | Cons Int Clist
            | Rec (Clist -> Clist)
```

Ejemplos de listas cíclicas con esta representación son las siguientes:

- La representación de la lista `[1,2,1,2,1,2,...` se observa en la figura 2.1

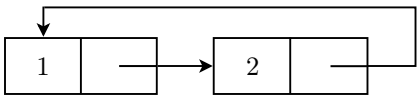


Figura 2.1: `clist1 = Rec (\xs -> Cons 1 (Cons 2 xs))`

- La representación de la lista `[1,2,3,2,3,2,3,...` se observa en la figura 2.2

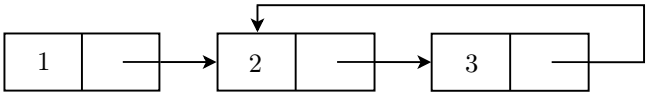


Figura 2.2: `clist2 = Cons 1 (Rec (\xs -> Cons 2 (Cons 3 xs)))`

Aunque los ciclos son explícitos en esta representación (basándonos en la intuición de que *Rec* significa fix), tenemos algunos problemas, el primero es que varias funciones que manipulan estructuras cíclicas necesitan desdoblar los ciclos, por ejemplo, al quitar el primer elemento de una lista cíclica puede ser que haya un apuntador a él o a una posición posterior, y se debe saber donde empezar a repetir para imprimir la lista resultante. Otra deficiencia es que se puede representar el ciclo vacío, el cual tampoco puede ser desenrollado:

```
empty = Rec (\xs -> xs)
```

Cabe mencionar que la representacion no es única. Por ejemplo, la lista cíclica `clist1` puede ser representada como:

```
clist1 = Rec (\xs -> Rec (\ys ->
         cons 1 (cons 2 (Rec \zs -> xs)))
```

Esencialmente, el constructor `Rec` etiqueta una posición en una lista con una variable que puede ser usada en el resto de la lista como un apuntador de regreso que forma un ciclo. Una lista contiene a lo más un ciclo, así que puede haber a lo más una posición de señalización (esto es porque las listas son lineales, en estructuras ramificadas puede haber más ciclos). Pero nada impide mas de una etiqueta para esa posición y etiquetas para otras posiciones.

También se pueden representar listas que no son cíclicas:

```
aciclica = Rec (\xs -> Cons 1 (cmap (+1) xs))
```

donde `cmap` representa a la función map de listas. Al aplicar el operador de punto fijo la lista va aumentado el valor de cada elemento, por ejemplo:

```
aciclica = [1,2,3,4,5,...]
```

El problema del ciclo vacío puede ser resuelto fácilmente. Para desactivarlo, es suficiente pedir que `Rec` siempre venga en combinación con `Cons`. Introducimos un nuevo constructor `Rcons` que combina ambos:

```
data Clist = Nil
           | Cons Int Clist
           | Rcons Int (Clist -> Clist)
```

Aparte del ciclo vacío, esta modificación evita múltiples etiquetas para una posición, sin embargo aún se puede elegir (para cada posición sin apuntador) si se debe etiquetar usando `Rcons` o simplemente utilizar `Cons`. Para quitar este tipo de ambigüedades debemos exigir un etiquetado único. De este modo llegamos a una definición simple:

```
data Clist = Nil
           | Rcons Int (Clist -> Clist)
```

Esta representación, en efecto, prohíbe el ciclo vacío y asegura que todas las listas cíclicas tienen una representación única.

Las preguntas a responder ahora son las siguientes:

- ¿Se puede definir un tipo de listas cíclicas que me asegure que cada una de sus instancias sea una lista cíclica?
- ¿Cómo se puede generar una biblioteca de estructuras cíclicas con sus respectivas funciones que tenga la posibilidad de manejar ciclos explícitamente y no tenga los problemas que enfrentamos al definirla con `fix` o `Rec`?

En el siguiente capítulo se presentan las estructuras cíclicas con tipos anidados para responder las preguntas anteriores.

Capítulo 3

Tipos anidados

A continuación se presentan estructuras cíclicas con tipos anidados, primero se realizará un comparativo con éstas y las estructuras cíclicas definidas con tipos regulares usando algunos ejemplos. Después se presentarán las listas y los árboles cíclicos, que son las estructuras de datos mas utilizadas en ciencias de la computación.

3.1. Tipos de datos regulares y anidados

En un **tipo de dato regular** recursivo, las llamadas del tipo de dato declarado en la derecha de la expresión son copias del lado izquierdo, por ejemplo:

- Listas de elementos de tipo `a`.

  ```
  data List a = Empty | Cons a (List a)
  ```

- Árboles binarios con elementos de tipo `a` excepto las hojas.

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

- Árboles generales con elementos de tipo `a`. Los nodos se representan con el constructor `Branch` y pueden tener cualquier número de subárboles a través de una lista de `GTree a`.

  ```
  data GTree a = Branch a (List (Gtree a))
  ```

En los ejemplos anteriores el parámetro de tipo *a* es igual en ambos lados.

Por el contrario, un **tipo de dato anidado**, también conocido como tipo de dato **heterogéneo** (o no regular), es aquel donde las ocurrencias del tipo de dato

declaradas a la derecha de la definición aparecen con diferentes instancias de los parámetros que lo definen, por lo que la recursión es anidada[2], a continuación se presentan algunos ejemplos:

```
data Perfect a = Zero a | Succ (Perfect (a,a))
```

```
data Tri a = Single a | Compp (Tri (Int,a))
```

En el primer tipo de dato el parámetro de tipo `a` cambia en la llamada recursiva por `(a,a)` y en el segundo ejemplo `a` cambia por `(Int,a)`. El tipo `Perfect` representa árboles binarios perfectos y `Tri` matrices triangulares, en las secciones 3.1.1 y 3.1.2 se habla con más detalle sobre ellos.

Existen tipos anidados en los que el parámetro de tipo en la llamada recursiva incluye al constructor del mismo tipo[2]:

```
data Bush a = NilB | ConsB a (Bush (Bush a))
```

A estos tipos de datos se les llama **realmente anidados**.

Ejemplos más sofisticados de tipos de datos anidados son los siguientes:

- *Términos del cálculo lambda con la notación de Bruijn*: este tipo permite incluir las variables libres de una expresión lambda en la notación de Bruijn, se pueden ver más detalles en [3].

  ```
  data Term v = Var v
              | App (Pair (Term v))
              | Lam (Term (Incr v))

  data Incr v = Zero | Succ v
  ```

- *Listas de acceso aleatorio*: *"son listas en las que las funciones de búsqueda y actualización son muy eficientes"*, se pueden ver más detalles en [22].

  ```
  data Ral a = Nil
             | Zero (Ral (a,a))
             | One a (Ral (a,a))
  ```

Para entender mejor el comportamiento de los parámetros de tipo en un tipo anidado, se define el tipo *Nest*.

```
data Nest a = NilN | ConsN (a,Nest (a,a))
```

Los elementos del tipo *Nest* son como listas, pero es claro que no son homogéneas, puesto que en cada llamada las entradas están duplicadas.

```
data Nest a = NilN | ConsN (a,Nest (a,a))

data Nest (a,a) = NilN | ConsN ((a,a),Nest ⟨(a,a),(a,a)⟩)

data Nest ⟨(a,a),(a,a)⟩ = NilN
                  | ConsN (⟨(a,a),(a,a)⟩,
                          Nest [⟨(a,a),(a,a)⟩,⟨(a,a),(a,a)⟩])

          ⋮                              ⋮
```

Enfocándonos sólo en el parámetro:

```
Nest a
Nest (a,a)
Nest ⟨(a,a),(a,a)⟩
Nest [⟨(a,a),(a,a)⟩,⟨(a,a),(a,a)⟩]
          ⋮
```

Dado que `a` es una variable de tipo, al sustituirla por un tipo arbitrario fijo tenemos una familia indexada por `a`.

No es posible definir un elemento de la familia por sí solo, por ejemplo, en el tipo `List a` definido previamente podemos sustituir el parámetro `a` por cualquier tipo, digamos `Char`, de este modo el tipo de listas de caracteres queda definido de manera única. En cambio, si se sustituye `a` por `Char` en el tipo anidado `Nest`, al declarar una estructura de tipo `Nest Char`, ésta puede tener una subestructura de tipo `Nest (Char,Char)` como se ve en este ejemplo:

```
nest1 :: Nest Char
nest1 =  ConsN ('a',
               ConsN ( ('b','c'),
                     ConsN ( (('d','e'),('f','g')), Nil)
                     )
               )

nest2 :: Nest (Char,Char)
nest2 = ConsN ( ('b','c'),
              ConsN( (('d','e'),('f','g')), Nil)
              )
```

`nest2` es subestructura de `nest1` y tienen tipos distintos.

Por lo anterior, debemos ser cuidadosos con la recursión en estructuras anidadas, pues se definen de manera distinta que en las regulares.

3.1.1. Comparativo entre tipos regulares y anidados

Cuando se implementan estructuras de datos con tipos anidados se debe tener cuidado con la definición de las funciones, en especial cuando se hace recursión. Para esclarecer lo anterior se dará la implementación de una estructura de ambas formas: la regular y la anidada. La estructura que se va a comparar es el árbol binario perfecto[1] con información sólo en las hojas.

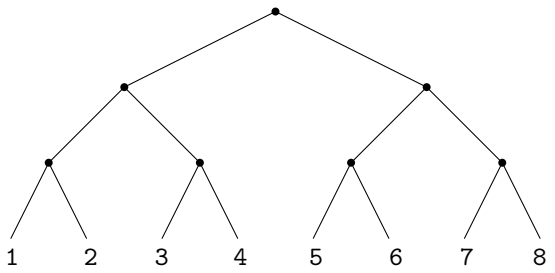


Figura 3.1: Ejemplo de árbol binario perfecto.

Implementación con tipo regular

La representación del árbol binario perfecto con tipo regular es la siguiente:

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

Los constructores son:

```
Leaf :: a  -> Tree a
Node :: Tree a -> Tree a -> Tree a
```

Un ejemplo de árbol binario perfecto con este tipo es el de la figura 3.2.

Cabe destacar que el árbol dado de ejemplo no es perfecto, por lo que se debe definir una función que verifique si un árbol es perfecto, sería así:

```
perfect :: Tree a -> Bool
perfect (Leaf x)       = True
perfect (Node t1 t2) = perfect t1 &&
                         perfect t2 &&
                         (hg t1 == hg t2)
```

[1] En un árbol binario perfecto todas las hojas estan a la misma profundidad (distancia desde la raíz).

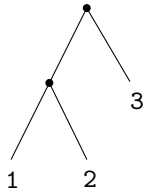


Figura 3.2: `Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)`

Algunas funciones útiles sobre este tipo de dato son las siguientes:

- Altura de un árbol:

```
hg :: Tree a -> Int
hg (Leaf x) = 0
hg (Node t1 t2) = 1 + max (hg t1) (hg t2)
```

- Aplanamiento de un árbol (regresa una lista con las hojas):

```
flat :: Tree a -> [a]
flat (Leaf x) = [x]
flat (Node t1 t2) = flat t1 ++ flat t2
```

- El map de un árbol (aplica una función a cada hoja):

```
mapt :: (a -> b) -> Tree a -> Tree b
mapt f (Leaf x) = Leaf (f x)
mapt f (Node t1 t2) = Node (mapt f t1) (mapt f t2)
```

Estas funciones utilizan el mecanismo usual de recursión en estructuras de datos funcionales.

Implementación con tipo anidado

La representación anidada de los árboles binarios perfectos con información en las hojas es la siguiente:

```
data Perfect a = Zero a | Succ (Perfect (a,a))
```

En donde un árbol perfecto tiene la información de las hojas en `Zero x` y la información de la altura del árbol en `Suc (a,a)`, por ejemplo, la representación del árbol perfecto de la figura 3.3 con el tipo regular es la siguiente:

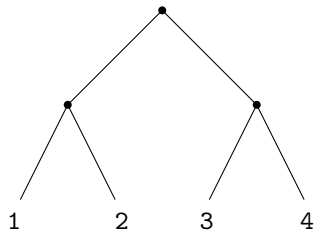


Figura 3.3: El árbol `perf1`

```
perf1 = Node (Node (Leaf 1) (Leaf 2))
             (Node (Leaf 3) (Leaf 4))
```

Y ese mismo árbol con el tipo anidado `Perfect Int` se representa de la siguiente manera:

```
perf1 = Succ (Succ (Zero ((1,2),(3,4))))
```

Al contrario de lo que sucede con el tipo regular, el tipo anidado de árboles perfectos asegura, en tiempo de compilación, que cualquier árbol definido por el programador es perfecto sin necesidad de una función que lo compruebe. Por lo anterior se dice que el tipo cumple un invariante estático.

En lenguajes como HASKELL o ML, se debe tener cuidado al definir funciones recursivamente, como por ejemplo la función altura (`ap`) en un árbol perfecto:

```
ap :: Perfect a -> Int
ap (Zero _) = 0
ap (Succ p) = (ap p) + 1
```

La llamada recursiva de `ap` colocada con letra cursiva exige una clase especial de recursión puesto que se está empleando otra instancia de la función definida, es decir, aplicar la llamada recursiva al parámetro `p` en `Succ p` es llamar a la función `hp :: Perfect (a,a) -> Int` porque el parámetro del constructor de `Succ` es `Perfect (a,a)`, esto se conoce como **recursión polimórfica**.

Sin embargo al escribir la función `ap` en el intérprete, ésta funciona sin problemas:

```
> ap (Succ (Succ (Zero ((1,2),(3,4)))))
   2
```

Para que el intérprete de Haskell no marque errores al trabajar con recursión polimórfica en la función `ap` se debe especificar su firma (`ap :: Perfect a -> Int`), de otro modo el intérprete marcaría error de tipo, pues en este caso la inferencia de tipos es indecidible.

Otro ejemplo es dado por la función `mapt`, que transforma árboles perfectos de acuerdo a una función dada.

```
mapt :: (a -> b) -> (Perfect a -> Perfect b)
mapt f (Zero x) = Zero (f x)
mapt f (Succ tp) = Succ (mapt (pairf f) tp)

pairf :: (a -> b) -> (a,a) -> (b,b)
pairf f (x,y) = (f x,f y)
```

Al igual que en el caso anterior, la llamada recursiva de `mapt` colocada con letra cursiva modifica su primera entrada con la función `pairf` de `(a -> b)` a `(a,a) -> (b,b)` para que en la llamada recursiva sea coherente con el tipo `Perfect (a,a) -> Perfect (b,b)`. Para que el intérprete no marque errores es necesario escribir la signatura de tipo. Ejemplo de evaluación:

```
> mapt (+1) (Succ (Succ (Succ (Zero (((1,2),(3,4)),((5,6),(7,8))))
             )))

     Succ (Succ (Succ (Zero (((2,3),(4,5)),((6,7),(8,9)))))))
```

La función `flatp` devuelve los elementos de un árbol perfecto en una lista. Aquí se encapsula la concatenación de las listas parciales en las llamadas recursivas con la función `fork` y la lista de un solo elemento con `wrap`.

```
flatp :: Perfect a -> [a]
flatp = flp wrap

-- encapsula un elemento en una lista
wrap :: a -> [a]
wrap x = [x]

-- cuando en la llamada recursiva aparace un par, se llama a la
-- función fork para que realice la concatenación
flp :: (a -> [b]) -> (Perfect a -> [b])
flp f (Zero x) = f x
flp f (Succ pt) = flp (fork f) pt

-- concatena los elementos de un par como listas
fork :: (a -> [b]) -> (a,a) -> [b]
fork f (x,y) = (f x) ++ (f y)
```

Ejemplo de evaluación:

```
> flatp (Succ (Succ (Zero ((1,2),(3,4)))))
    [1,2,3,4]
```

Las funciones recurrentes con tipos anidados no siempre se pueden manejar con recursión polimórfica, como se verá en las secciónes 3.2 y 3.3 con las listas y árboles cíclicos, respectivamente.

3.1.2. Matrices Triangulares

Un ejemplo de estructura infinita que se define con tipos anidados es el de las *matrices triangulares*, que son matrices cuadradas de enteros sin información debajo de la diagonal. De manera equivalente, se pueden definir como matrices simétricas en las que la información redundante debajo de la diagonal se omite.

Los elementos en la diagonal juegan un papel distinto de los otros. En la construcción de la estructura se define un tipo E fijo de objetos que no están en la diagonal y un tipo parámetro A de objetos de la diagonal. Por ejemplo, en una matriz numérica, se podría requerir que los elementos enteros de la diagonal fueran distintos de cero para que fuera invertible, o cumplieran otra propiedad que diferencie su tipo de los que no están en la diagonal, como sucede en la siguiente matriz:

$$\begin{array}{ccccc} 2 & -1 & 8 & 7 & 11 \\ & 7 & 23 & 13 & -7 \\ & & 9 & 8 & 20 \\ & & & 3 & 2 \\ & & & & 6 \end{array}$$

Implementación no anidada

Las dos formas usuales de definir matrices en Haskell son las siguientes:

- *Listas de listas*

  En este caso conviene utilizar sinónimos de tipo, primero para ver a las listas como vectores y a las listas de vectores como matrices:

  ```
  type Vector  = [Int]

  type Matriz = [Vector]
  ```

  Una limitante de esta representación es que no podemos definir matrices de dos tipos de datos diferentes, la declaración anterior define matrices finitas de enteros, nuesto tipo de matrices triangulares se limitaría a un solo tipo y no a dos (E y A) como queremos.

```
matriz         = [[1,5,3,8],[-4,9,2,2],[6,6,7,8],[6,-9,1,6]]
mat_triangular = [[2,-1,8,7],[7,23,13],[9,8],[3]]
```

Como ocurre con los árboles binarios perfectos de la sección 3.1.1, se debe definir una función que verifique que la matriz definida de esta manera sea triangular.

- *Tipo de dato Array*

  Array es un tipo de dato predefinido de Haskell para manejar arreglos de una o varias dimensiones (ver más detalles en [32]). Un *Array* está formado por dos componentes:

  - `(i,i)`
    Un par de límites, cada uno del tipo de índice de arreglo. Esos límites son los índices menores y mayores del arreglo. Por ejemplo un vector de longitud diez tiene límites `(1,10)` y una matriz de diez por diez tiene límites `((1,1),(10,10))`.
  - `[(i, e)]`
    Una lista de pares (o lista de asociaciones) de la forma *(indice, valor)*. Una asociación `(i,x)` indica que `x` es el valor del *Array* en el índice `i`.

  Ejemplos de *Array* unidimensional y bidimensional:

```
vect = array (1,3) [(1,10),(2,20), (3,30)]
                        -- representa al vector [10,20,30]

triangular = array ((1,1),(4,4))
              [ ((1,1), 2),((1,2), -1),((1,3), 8),((1,4), 7),
                ((2,2), 7),((2,3), 23),((2,4),13),
                ((3,3), 9),((3,4),  8),
                ((4,4), 6) ]

                            2 -1  8  7
                               7 23 13
                                  9  8
 -- representa a la matriz            6
```

  Un *Array* es una función parcial del conjunto de índices al conjunto de valores, esto se refleja en el segundo ejemplo con la matriz `triangular` que queda indefinida en en `(2,1)`, `(3,1)`, `(3,2)`, `(4,1)`, `(4,2)` y `(4,3)`. Para este tipo de funciones, el acceso a los valores se realiza con auxilio del operador `(!)`:

```
> triangular ! (1,3)
   8

> triangular ! (4,4)
   6
```

La facilidad de esta representación es que podemos dar en desorden (respecto al índice) la lista de pares del *Array* así como definirla con listas por comprensión. Sin embargo esta limitada a un solo tipo como ocurre con las listas de listas y también se debe implementar un función que indique si una matriz es triangular.

Implementación anidada

Ahora se definirá a las matrices triangulares con tipos anidados, ésta parte se basa en el artículo de Ralph Matthes y Martin Strecker *"Verification of the redecoration algorithm for triangular matrices"*[20].

Si A es el tipo de los elementos de la diagonal, entonces *Tri A* denota el tipo de las matrices triangulares con elementos de tipo A en la diagonal y elementos de tipo E (fijo) fuera de ella. Esta representación no tiene triángulos vacíos, el menor elemento de *Tri A* contiene un solo elemento que pertenece a la diagonal y por tanto es de tipo A. Lo anterior se visualiza en el constructor:

$$single : A \to Tri\ A$$

El caso general puede visualizarse así:

$$\begin{array}{r|l} A & E\ E\ E\ E \\ & A\ E\ E\ E \\ & \quad A\ E\ E \\ & \quad\ \ A\ E \\ & \quad\ \ \ A \\ & \quad\ \ \ \ldots \end{array}$$

La linea vertical corta el triangulo en un elemento de tipo A y un "trapecio", con la fila superior consistente únicamente de elementos de tipo E. Con el propósito de tener un proceso inductivo, se realizará el apareamiento de cada elemento $M_{i-1,i}$ con $M_{i,i}$ (indicados abajo con letra negrita) y los trapecios estarán en correspondencia uno a uno con los triángulos:

$$\begin{array}{l} \boldsymbol{E}\ E\ E\ E \\ \boldsymbol{A}\ \boldsymbol{E}\ E\ E \\ \quad \boldsymbol{A}\ \boldsymbol{E}\ E \\ \quad\ \ \boldsymbol{A}\ \boldsymbol{E} \\ \quad\ \ \ \boldsymbol{A} \\ \quad\ \ \ \ldots \end{array} \longrightarrow \begin{array}{llll} E \times A & E & E & E \\ & E \times A & E & E \\ & & E \times A & E \\ & & & E \times A \end{array} \quad \ldots$$

El segundo constructor es una vista del trapecio como triángulo:

$$cons : A \to Tri\ (E \times A) \to Tri\ A$$

de nuevo con A una variable de tipo. Por tanto los triángulos no triviales consisten del elemento superior de la izquierda, tomado de A, y un triángulo con elementos en la diagonal tomados de $E \times A$.

La implementación en Haskell de matrices triangulares con elementos de tipo `a` en la diagonal y enteros fuera de ella, es la siguiente:

```
data Tri a = Single a
           | Cons a (Tri (Int,a))
```

Ejemplos:

- ```
  tri1 :: Tri String
  tri1 = Cons "cad" (Single (2,"dac"))
  ```
  y representa a la matriz:

$$\begin{array}{cc} cad & 2 \\ & dac \end{array}$$

- ```
  tri2 :: Tri Int
  tri2 = Cons 20 (Cons (2,30) (Single (3,(4,40))))
  ```
  y representa a:

$$\begin{array}{ccc} 20 & 2 & 3 \\ & 30 & 4 \\ & & 40 \end{array}$$

- ```
  tri3 :: Tri Char
  tri3 = Cons 's' (Cons ( 2,  't')
                  (Cons ( 3, ( 4,  'u'))
                  (Cons ( 5, ( 6, ( 7,  'v')))
                  (Cons ( 8, ( 9, (10, (11,  'w'))))
                 (Single (12, (13, (14, (15, (16, 'x'))))))))))
  ```
  y representa a:

$$\begin{array}{cccccc} s & 2 & 3 & 5 & 8 & 12 \\ & t & 4 & 6 & 9 & 13 \\ & & u & 7 & 10 & 14 \\ & & & v & 11 & 15 \\ & & & & w & 16 \\ & & & & & x \end{array}$$
  ```

Para hacer la matriz triangular infinita se quita el constructor `Single`

```
data Tri a = Cons a (Tri (Int,a))
```

Como ejemplos de funciones con esta estructura tenemos a `top`, la cual devuelve el primer elemento de la matriz (superior izquierdo):

```
top :: Tri a -> a
top (Single a) = a
top (Cons a b) = a
```

Un ejemplo de función mucho más interesante es la de sustitución, que en [20] se le llama *redecoración*; consiste en reemplazar los elementos de la diagonal con otros de un tipo distinto. En cuanto a la implementación, el enfoque dado en [20] es el siguiente: dados dos tipos `a` y `b` y una función `f :: Tri a -> b` (regla de redecoración), definimos la función `redec f :: Tri a -> Tri b`. La idea intuitiva de redecoración en el caso de triángulos es ir recursivamente a través del triángulo y sustituir cada elemento de la diagonal por el resultado de aplicar `f` al subtriángulo que se extiende a la derecha, debajo del renglón del elemento diagonal. Para lo anterior, primero se define la función `cut` que quita el primer renglón de un trapecio (matriz de tipo `Tri (Int,a)`) para así regresar una matriz de tipo `Tri a`:

```
cut :: Tri (Int,a) -> Tri a
cut (Single (e,a)) = Single a
cut (Cons (e,a) r) = Cons a (cut r)
```

Dado que `Tri (Int,a)` es anidado, se debe resaltar que la aparición de `r` en el lado izquierdo de la definición (marcada con letra cursiva) es de tipo `Tri (Int,Int,a)` mientras que del lado derecho tiene tipo `Tri (Int,a)`; por lo que, al igual que en la sección 3.1.1 ocurría con algunas funciones del tipo `Perfect`, se está utilizando recursión polimórfica y es necesario dar la signatura de tipo `cut :: Tri (Int,a) -> Tri a` para que el intérprete no marque errores.

Cuando la redecoración de un triángulo haga la llamada recursiva con `Tri (Int,a)` (que es la representación de trapecio de una matriz triangular), se debe hacer un ajuste en la función de redecoración `Tri a -> b` que ahora necesita ser de tipo `Tri (Int,a) -> (Int,b)`, esto se logra fácilmente con el uso de las funciones `top` y `cut`:

```
lift  :: (Tri a -> b) -> Tri (Int,a) -> (Int,b)
lift f r = (fst (top r), f (cut r))
```

La función `lift` regresa una tupla que tiene al entero que está del lado superior izquierdo del trapecio y al resultado de aplicar la redecoración al triángulo que se obtiene al quitar el primer renglón del trapecio.

Con lo visto previamente, estamos en condiciones para definir la función de redecoración en matrices triangulares:

```
redec :: (Tri a -> b) -> Tri a -> Tri b
redec f (Single x) = Single (f (Single x))
redec f (Cons a r) = Cons (f (Cons a r)) (redec (lift f) r)
```

que de nuevo utiliza recursión polimórfica al cambiar `f` por `lift f` del lado derecho de la función. Por ejemplo, aplicar la siguiente redecoración:

```
paso :: Tri Char -> Int
paso (Single x) = ord x
paso (Cons a r) = ord a
```

al triángulo `tri3 :: Tri Char` definido anteriormente en los ejemplos, nos da lo siguiente:

```
> redec paso tri3
     Cons 115 (Cons   ( 2,  116)
              (Cons   ( 3, (  4,  117))
              (Cons   ( 5, (  6, (  7,  118)))
              (Cons   ( 8, (  9, ( 10, ( 11,  119))))
            (Single   (12, ( 13, ( 14, ( 15, ( 16, 120)))))))))))
```

que representa la matriz:

$$\begin{array}{cccccc}
\mathit{115} & 2 & 3 & 5 & 8 & 12 \\
 & \mathit{116} & 4 & 6 & 9 & 13 \\
 & & \mathit{117} & 7 & 10 & 14 \\
 & & & \mathit{118} & 11 & 15 \\
 & & & & \mathit{119} & 16 \\
 & & & & & \mathit{120}
\end{array}$$

La redecoración consistió en cambiar los caracteres de la diagonal por sus representaciones enteras con la función `ord` predefinida en Haskell[33].

Los tipos de datos inductivos, coinductivos y sus constructores asociados; en particular, los esquemas de recursión estructurada (*fold*) y correcursión (*unfold*), recursión primitiva, entre otros, son importantes para los lenguajes de programación funcional y han sido formalmente estudiados (ver [18]).

Pero cuando nos enfrentamos al reto de definir estructuras cíclicas, las cuales son comúnes en programación ¿cómo planteamos una estructura de dato cíclica que sea manipulable a nivel funcional? En el paradigma de programación imperativa es común usar apuntadores para representar ciclos, sin embargo en programación funcional se utilizan marcadores de repetición en las estructuras[2].

[2]De aquí en adelante se seguirán nombrando "apuntadores".

A continuación se desarrolla un sistema para representar estructuras cíclicas con tipos anidados, en el que no es necesario definir funciones que comprueben si una estructura es cíclica.

3.2. Listas cíclicas

Anteriormente se definieron estructuras cíclicas en HASKELL con ciertas desventajas. Si queremos una representación de listas cíclicas que evite el uso de tipos de orden superior (como se expuso en la sección 2.3) y genere ciclos manipulables, en [8] se propone poner apuntadores hacia alguna de sus posiciones de la siguiente manera:

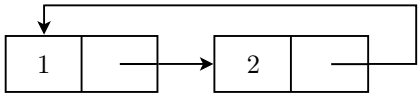
```
data Clist a = Point a
             | Rcons Int (Clist (Lift a))

data Lift a = Zero | Suc a
```

El constructor `Point` no representa una lista, sino un apuntador, que indica el inicio de la lista (`Zero`) o alguna posición más adelante (`Suc p`). El tipo `Clist` es claramente anidado, porque el parámetro `a` cambia en la llamada recursiva por `Lift a`.

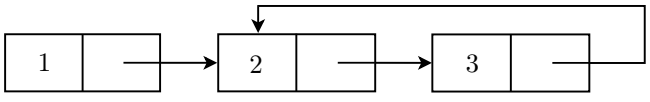
Ejemplos:

- `clist1 = Rcons 1 (Rcons 2 (Point Zero))`



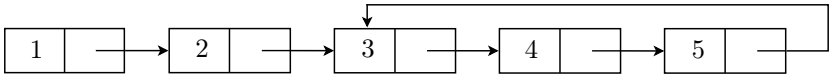
`[1,2,1,2,1,2,...]`

- `clist2 = Rcons 1 (Rcons 2`
  `(Rcons 3 (Point (Suc Zero))))`



`[1,2,3,2,3,2,3,...]`

- `clist3 = Rcons 1 (Rcons 2 (Rcons 3 (Rcons 4 (Rcons 5 (Point (Suc (Suc (Zero))))))))`



`[1,2,3,4,5,3,4,5,3,4,5,...]`

Cabe mencionar que un tipo anidado es un tipo recursivo no regular en el que el parámetro de tipo cambia en la llamada recursiva. `Clist a` es anidado y define listas cíclicas, pero ¿aquellas listas son de enteros?. Una peculiaridad de su parámetro de tipo (`a`) es que no define el tipo de los elementos de la lista.

El parámetro de tipo `a` en `Clist a` permite manejar los apuntadores, ya que al cambiar por `Lift a`, accede a los constructores `Zero` y `Suc a` que actúan como numerales para indicar la posición a la que se va a regresar en la lista. Sabemos que la lista es de enteros porque se establece explícitamente en el constructor `Rcons`.

Es posible también definir esta estructura de lista cíclica para elementos de cualquier tipo, agregando un parámetro de tipo no anidado:

```
data Clist a b = Point a
               | Rcons b (Clist (Lift a) b)

data Lift a = Zero | Suc a
```

Por simplicidad en la representación de las siguientes funciones, seguiremos usando el tipo de listas de enteros.

Funciones constructoras

Para construir una lista a partir de una lista y un entero (en un tipo de dato regular) se utiliza el constructor `Cons`.

```
Cons :: Int -> List Int -> List Int

lista1 = Cons 0 Nil           -- lista1 = [0]

lista2 = Cons 2 lista1        -- lista2 = [2,0]
```

Sin embargo en nuestro tipo anidado no podemos agregar un entero a una lista cíclica de la misma forma porque el apuntador cambia de posición. Por ejemplo:

```
clist1 = Rcons 1 (Point Zero)         -- clist1 = [1,1,1,...]

clist2 = Rcons 2 clist1
```

```
clist2 = Rcons 2 (Rcons 1 (Point Zero))
```

es la lista

`[2,1,2,1,2,1,...]`

pero la intención era construir la lista

`[2,1,1,1,...]`

cuya representación es:

`Rcons 2 (Rcons 1 (Point (Suc Zero)))`

Por eso se debe implementar una función que ajuste los apuntadores cuando agregamos un entero en la cabeza de la lista.

La función `ccons` construye una lista cíclica a partir de un entero y una lista cíclica.

```
ccons :: Int -> Clist a -> Clist a
ccons x xs = Rcons x (shift xs)

shift :: Clist a -> Clist (Lift a)
shift (Point z) = Point (Suc z)
shift (Rcons x xs) = Rcons x (shift xs)
```

`shift` es la función de ajuste de los apuntadores, visita recursivamente cada nodo de la lista sin realizar modificaciones y cuando encuentra un apuntador a determinada posición (`Point z`), lo incrementa (`Point (Suc z)`) para conservar el ciclo que se tenía previamente, dado que se ha agregado un nuevo nodo a la lista cíclica.

Ejemplo de evaluación de `ccons`:

```
cyclic_1 = Rcons 4 (Rcons 7 (Rcons 9 (Point Zero)))
                    -- [4,7,9,4,7,9,...]
ccons 2 cyclic_1 = Rcons 2 (shift cyclic_1)
      = Rcons 2 (shift (Rcons 4 (Rcons 7 (Rcons 9 (Point Zero)))))
      ⇝ Rcons 2 (Rcons 4 (shift (Rcons 7 (Rcons9 (Point Zero)))))
      ⇝ Rcons 2 (Rcons 4 (Rcons 7 (shift (Rcons 9 (Point Zero)))))
      ⇝ Rcons 2 (Rcons 4 (Rcons 7 (Rcons 9 (shift (Point Zero)))))
      ⇝ Rcons 2 (Rcons 4 (Rcons 7 (Rcons 9 (Point (Suc Zero)))))
                    -- [2,4,7,9,4,7,9,...]
```

Funciones destructoras (correcursivas)

Dado que las listas son infinitas se debe aplicar funciones correcursivas definidas a partir de los destructores de la lista[3], es decir, las funciones que obtienen la cabeza y la cola. La función que obtiene la cabeza (`chead`) de una lista es sencilla.

```
chead :: Clist a -> Int
chead (Point z) = error "no es una lista"
chead (Rcons x _) = x
```

Basta con definir `chead` sólo para el constructor `Rcons` porque en la definición del tipo `Clist` se explica que `Point` no representa una lista; sin embargo, se define `chead` para apuntadores por buena práctica de programación, es decir, para cada constructor de `Clist`, evitando así errores en tiempo de ejecución.

Pero para la cola se tiene el mismo problema que se presentó con la función que contruye listas cíclicas (donde se utilizó `shift`), ya que en la expresión `Rcons x xs` se tiene que `xs` tiene tipo `Clist (Lift a)` y no `Clist a` como se espera.

Por eso se define una función adecuada que construya la cola de una lista cíclica.

```
ctail :: Clist a -> Clist a
ctail (Point z) = error "no es una lista"
ctail (Rcons x xs) = csnoc x xs

csnoc :: Int -> Clist (Lift a) -> Clist a
csnoc y (Point Zero) = Rcons y (Point Zero)
csnoc y (Point (Suc z)) = Point z
csnoc y (Rcons x xs) = Rcons x (csnoc y xs)
```

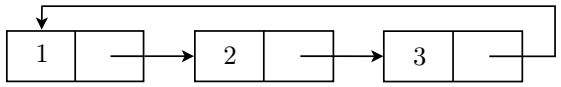
El caso de la cola de un apuntador se define con `error` por buena práctica de programación.

Si no es un constructor `Point`, `ctail` llama a la función `csnoc`, la cual recorre recursivamente la cabeza de la lista a la derecha hasta llegar al apuntador, en cuyo caso, la permanencia de la cabeza en esa posición se decide dependiento del mismo apuntador, se tienen dos casos:

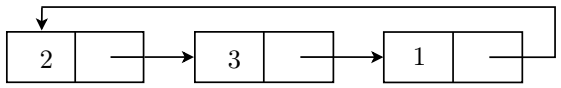
1: Si es un apuntador al primer elemento (`Point Zero`), la cabeza se queda en esa posición.

Ejemplo de evaluación:

[3]La correcursión es el dual de la recursión para estructuras infinitas. Para observar cómo se comporta una estructura infinita hay que destruirla en sus partes finitas. Por ejemplo una lista infinita es un objeto que no puede ser visto en su totalidad, sólo podemos conocer individualmente sus partes finitas[10].




```
cyclic_1 = Rcons 1 (Rcons 2 (Rcons 3 (Point Zero)))
```




```
ctail cyclic_1 = Rcons 2 (Rcons 3 (Rcons 1 (Point Zero)))
```

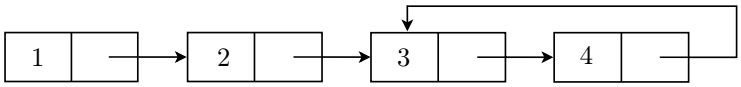
El procedimiento paso a paso se visualiza de la siguiente forma:

```
ctail {Rcons 1 (Rcons 2 (Rcons 3 (Point Zero)))}
    = csnoc 1 (Rcons 2 (Rcons 3 (Point Zero)))
    ⇝ Rcons 2 (csnoc 1 (Rcons 3 (Point Zero)))
    ⇝ Rcons 2 (Rcons 3 (csnoc 1 (Point Zero)))
    ⇝ Rcons 2 (Rcons 3 (Rcons 1 (Point Zero)))
```

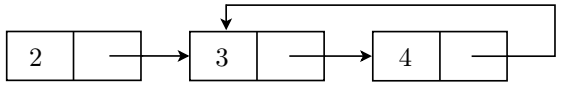
En este caso la cabeza pertenecía al ciclo, eso quiere decir que se repite un número infinito de veces en la cola, por eso se pasa de la posición inicial que ocupa en el periodo a la posición final del mismo.

2: Si es un apuntador a un nodo subsecuente, se elimina la cabeza y ese apuntador se recorre una posición a la derecha (pues ya se quitó el primer elemento).

Ejemplo de evaluación:




```
cyclic_2 = Rcons 1 (Rcons 2 (Rcons 3 (Rcons 4
         (Point (Suc (Suc Zero))))))
```




```
ctail cyclic_2 = Rcons 2 (Rcons 3 (Rcons 4
           (Point (Suc Zero))))
```

El procedimiento paso a paso se visualiza de la siguiente forma:

```
ctail cyclic_2

    = csnoc 1 (Rcons 2 (Rcons 3 (Rcons 4 (Point (Suc
                                    (Suc Zero))))))

    ⇝ Rcons 2 (csnoc 1 (Rcons 3 (Rcons 4 (Point (Suc
                                    (Suc Zero))))))

    ⇝ Rcons 2 (Rcons 3 (csnoc 1 (Rcons 4 (Point (Suc
                                    (Suc Zero))))))

    ⇝ Rcons 2 (Rcons 3 (Rcons 4 (csnoc 1 (Point (Suc
                                    (Suc Zero))))))

    ⇝ Rcons 2 (Rcons 3 (Rcons 4 (Point (Suc Zero))))
```

En este caso la cabeza no pertenecía al ciclo, por eso se elimina.

La función `ctail` actúa de manera opuesta a `ccons`:

- `ccons` a través de `shift` hace el ajuste del ciclo cuando se agrega un nodo, el ajuste es a la derecha, de `Point z` a `Point (Suc z)`
- `ctail` a través de `csnoc` hace el ajuste del ciclo cuando se quita un nodo, el ajuste es a la izquierda, de `Point (Suc z)` a `Point z`. De ahí que el nombre "csnoc" se haya elegido mediante la inversión del nombre "ccons".

Una vez resuelto el problema de construir y descomponer listas cíclicas, es necesario definir más funciones útiles sobre ellas. Muchas funciones usuales sobre listas implican visitar sus elementos y aplicarles alguna operación, como por ejemplo `map`. A continuación se presenta una forma de recorrer las listas cíclicas.

Unfold

Una lista cíclica es infinita y por tanto es una estructura coalgebraica, sin embargo se debe tener cuidado al definir el operador *unfoldr* porque no se hace del mismo modo que con las listas finitas.

Una lista coalgebraica es esencialmente una máquina de estados abstracta y el *unfoldr* de la lista coalgebraica calcula el rastreo de salida posiblemente infinito de dicha máquina[8] , esto quiere decir que al sacar la cabeza de nuestra lista coalgebraica se deja a la máquina en un estado S_1 que determina la cola

de la misma, y al volver a sacar la cabeza en S_1 se obtiene un estado diferente S_2. En una lista cíclica se repite un estado en algún momento[10].

Antes de continuar, recordemos la primera de las definiciones de `unfoldr` con listas finitas en la sección 1.4.2:

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f u = case f u of
Nothing -> [ ]
Just (x,v) -> x : (unfoldr f v)
```

donde `f :: b -> Maybe (a,b)` es una función de opción que analiza el valor de inicio (o semilla) `b` y lo **modifica** en cada paso recursivo, a la vez que va construyendo una lista mientras no se cumpla la condicion de terminación.

Siguiendo esta idea, si la condición de terminación no se cumple y alguno de los valores producidos se repite, la lista producida será infinita, pero periódica (o cíclica). Sin embargo la definición de lista cíclica `Clist a` dada al inicio de esta sección, obliga el uso de apuntadores con la presencia del constructor `Point a`, por lo que el uso de `unfoldr` de acuerdo a [8] no sigue una idea parecida a la explicada en la sección 1.4.2 y en lugar de comenzar con un valor de inicio arbitrario (por ejemplo un entero) se empieza con **la misma lista cíclica**; a la que en cada paso recursivo se le sacará la cabeza y la cola sera el estado (de la lista vista como máquina) en ese momento.

La idea para detectar estados repetidos es guardarlos en una lista y detectar si alguno se repite cada vez que se hace la llamada recursiva. Esto es posible si el conjunto de estados (listas cíclicas) tiene una relación de igualdad decidible (lo que se ve reflejado en la restricción de estas funciones con la clase de tipo[4] `Eq`). También se debe crear una lista con el estado del apuntador en cada paso recursivo, para que cuando se detenga la recursión al detectar la primera repetición, se coloque el apuntador adecuado en el final de la lista cíclica generada. A continuación se presenta la función `cunfoldL` que sigue esta idea.

```
cunfoldL :: Eq c => (c -> Maybe (Int,c)) -> c -> Clist a
cunfoldL = cunfoldL' [ ] [ ]

cunfoldL' :: Eq c => [c] -> [a] -> (c -> Maybe (Int,c)) -> c
             -> Clist a
cunfoldL' lc la ht c = case lookup c (zip lc la) of
  Nothing -> case ht c of
```

[4] `cunfoldL` y varias funciones definidas en lo subsecuente utilizan *contextos* y *clases de tipo*, los contextos se colocan delante de las expresiones de tipo e indican una restricción sobre él. La clase de tipo `Eq` define la igualdad `(==) :: a -> a -> Bool` y la desigualdad `(/=) :: a -> a -> Bool`, por ejemplo:

```
fun :: (Eq a) => a -> b -> (b,a)
```

se debe leer: "Para cada tipo `a` que es instancia de la clase `Eq`, `fun` tiene tipo `a -> b -> (b,a)`", en otras palabras, debe haber una igualdad definida en el tipo `a` para la correcta evaluación de `fun`. [16]

```
    Nothing -> error "nothing"
    Just (x,c') -> let lc' = lc ++ [c];
                       la' = Zero : map Suc la
                   in Rcons x (cunfoldL' lc' la' ht c')
  Just a -> Point a
```

Las funciones y constructores auxiliares de `cunfoldL` son los siguientes:

- `zip` toma dos listas y regresa el apareamiento uno a uno (en forma de tuplas) de los elementos de las dos listas.

  ```
  zip :: [a] -> [b] -> [(a,b)]
  zip (x:xs) (y:ys) = (x,y) : zip xs ys
  zip  xs     ys    = [ ]
  ```

- `lookup` busca una llave de tipo `a` en una lista de asociación y regresa el elemento al que está asociada la llave (de tipo `b`). Se utiliza el tipo `Maybe` en caso de que la búsqueda no sea exitosa.

  ```
  lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
  lookup  _        [ ]          =  Nothing
  lookup  key ((x,y):xys)
      | key == x          =  Just y
      | otherwise         =  lookup key xys
  ```

  Como es necesario comparar, se pide que el parámetro de tipo `a` tenga una definición de igualdad, es decir que sea comparable. En `cunfoldL`, a esta función se le pasará como segundo argumento una lista de pares generados previamente con `zip`, donde se hará la búsqueda.

Explicación de `cunfoldL`

`cunfoldL` es la aplicación parcial de `cunfoldL'` a dos listas vacías, en las que se guardarán los estados visitados y los apuntadores:

```
cunfoldL :: Eq c => (c -> Maybe (Int,c)) -> c -> Clist a
cunfoldL = cunfoldL' [ ] [ ]
```

La función `cunfoldL'` busca el valor inicial en la lista de estados

```
cunfoldL' lc la ht c = case lookup c (zip lc la) of
```

en el cual tenemos dos casos

1:
```
Nothing -> case ht c of
     Nothing -> error "nothing"
     Just (x,c') -> let lc' = lc ++ [c];
                        la' = Zero : map Suc la
                    in Rcons x (cunfoldL' lc' la' ht c')
```

El caso número uno siempre se ejecuta al inicio porque las listas están vacías.Lo que hace es evaluar la función de opcion (`ht`) en el valor de inicio `c` y aplica la llamada recursiva de `cunfoldL'` a las listas de los estados y los apuntadores actualizados.

La evaluación de la función de opción se detendrá pues se está trabajando con listas potencialmente infinitas, por lo cual no se llegará al caso en que devuelva `Nothing` (si fuera así se enviará un `error`). Sin embargo `cunfoldL'` regresa una lista cíclica que tiene un apuntador, el cual se determina en el siguiente caso.

2: `Just a -> Point a`

Si `lookup` encontró un estado repetido significa que ya llegó la señal para colocar el apuntador y la evaluación finalizará. De la lista de apuntadores se pone el correspondiente al estado repetido.

La función de paso que utiliza `cunfoldL` para listas cíclicas dadas como valores iniciales es `cheadtail`:

```
cheadtail :: Clist a -> Maybe (Int,Clist a)
cheadtail (Point z) = Nothing
cheadtail xs = Just (chead xs,ctail xs)
```

la cual devuelve un tipo `Maybe (Int, Clist a)`, la tupla contiene la cabeza y la cola de la lista cíclica.

Algunos ejemplos útiles bajo este esquema son los siguientes:

- `cmap` aplica una función a los elementos de una lista cíclica.

  ```
  cmap :: Eq (Clist a) => (Int -> Int) -> Clist a -> Clist b
  cmap f xs = cunfoldL ht xs where
        ht xs = case cheadtail xs of
                    Nothing -> Nothing
                    Just (x,xs') -> Just (f x,xs')
  ```

  La función de opción para `cunfoldL` es una sencilla modificación a `cheadtail`, en la que al separar una lista cíclica en cabeza y cola, le aplica una función a la cabeza.

  Ejemplo:

```
clist2 = Rcons 1 (Rcons 2 (Rcons 3 (Point (Suc Zero))))
```

es la lista `[1,2,3,2,3,2,3,2,3,...]`

```
> cmap (*2) clist2
      Rcons 2 (Rcons 4 (Rcons 6 (Point (Suc Zero))))
```

y el resultado como lista finita es `[2,4,6,4,6,4,6,4,6,...]`

- `czipWith` recibe una función de enteros, dos listas cíclicas y regresa una lista cíclica, resultante de aplicar la función a cada par de elementos de las dos listas, apareadas como en `zip`. En forma parecida a la de la función anterior, la semilla para `cunfoldL` es la pareja de listas cíclicas.

```
czipWith :: Eq (Clist a,Clist b) => (Int -> Int -> Int)
           -> Clist a -> Clist b -> Clist c
czipWith f xs ys = cunfoldL ht (xs,ys) where
     ht (xs,ys) = case cheadtail xs of
            Nothing -> Nothing
            Just (x,xs') -> case cheadtail ys of
                  Nothing -> Nothing
                  Just (y,ys') -> Just (f x y,(xs',ys'))
```

De nuevo la función de opción para `cunfoldL` es una modificación de `cheadtail`, ahora se obtienen las cabezas de las dos listas cíclicas con dos expresiones `case` anidadas y se les aplica la función dada.

Ejemplo:

```
clist3 = Rcons 1 (Rcons 2 (Rcons 3 (Point (Suc Zero))))
                            -- [1,2,3,2,3,2,3,2,3,...]

clist4 = Rcons 1 (Rcons 2 (Point (Suc Zero)))
                            -- [1,2,2,2,2,2,2,2,2,...]

> czipWith (+) clist3 clist4
      Rcons 2 (Rcons 4 (Rcons 5 (Point (Suc Zero))))
```

y el resultado como lista finita es `[2,4,5,4,5,4,5,4,5,4,5,,...]`

Despliegue de listas cíclicas

La manera más sencilla de desplegar una lista cíclica como lista de Haskell es la siguiente:

```
unwind :: Clist a -> [Int]
unwind xs = chead xs : unwind (ctail xs)
```

`unwind` también se puede definir de manera más simple con `cheadtail` y `unfoldr`:

```
unwind = unfoldr cheadtail
```

Como se vió en la sección 1.4.2, `unfoldr` es el dual del operador de plegado de listas regulares. La función `unwind` es la definición estándar de despliegue (o *show*) de la estructura de listas cíclicas aquí presentada.

Una lista cíclica es una representación finita de una lista infinita, semejante a la usada en aritmética para representar números racionales en notación decimal, por ejemplo $0{,}4\overline{285} = 0{,}4285285285....$ Por lo anterior, `unfoldr cheadtail` es un caso en el que unfold "desdobla" la representacion finita para obtener una lista infinita.

La función sobre listas regulares `append` no puede definirse para listas cíclicas. Cuando se trata de dos listas cíclicas es claro porque se requiere conocer el final de una de ellas; pero surge la pregunta acerca de si es posible añadir una lista cíclica a una finita. La respuesta es negativa ya que una lista finita no puede representarse con el tipo `Clist a` ya que cumple el invariante de lista cíclica y así las dos listas, la finita y la cíclica, diferirían en sus tipos. Tampoco es posible definir la función `reverse` porque una lista cíclica es infinita y `reverse` requiere el final de la lista.

En esta sección se ha dado una definición de listas cíclicas con tipos de datos anidados y funciones que muestran su utilidad. Ahora se presentan estructuras no lineales también con tipos anidados, mostrando mejor el poder de estas definiciones.

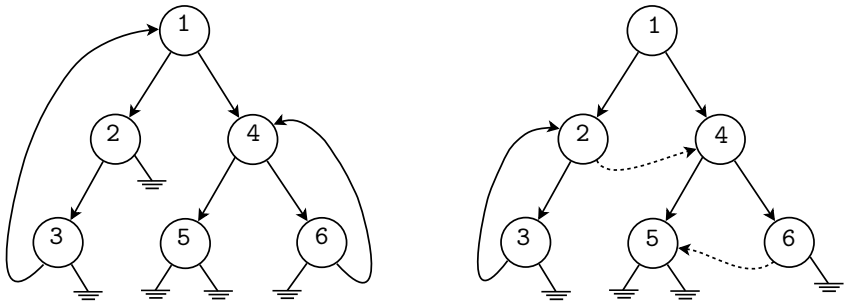
3.3. Árboles cíclicos

A continuación se presenta un tipo de árboles binarios cíclicos, de enteros y sin información en las hojas, con el uso de tipos anidados. La idea es análoga a la de las listas cíclicas; pero con esta estructura se pretende exponer la utilidad de los tipos anidados así como los retos a los que nos enfrentamos cuando definimos estructuras más complicadas.

Es relevante observar que en los árboles binarios cíclicos presentados en esta sección, los apuntadores que generan ciclos van de un nodo a cualquiera de sus ancestros ubicados en el camino de él a la raíz del árbol. Esta representación no contempla apuntadores a nodos que se encuentran fuera de ese camino, ese caso se conoce como *sharing*[11]. En las figura 3.4 se ejemplifica la diferencia entre un árbol cíclico como el que se presentará en esta sección y un árbol con *sharing*.

Una vez realizada la observación, procedamos con la definición del tipo de árboles binarios cíclicos sin información en las hojas el cual se llamará `Ctree` y esta basado en [8].

Se define un contructor `Pt` para los apuntadores, agregamos un construc-



sin *sharing* con *sharing*

Figura 3.4: Diferencia de árboles cíclicos

tor `Leaf` para indicar que se ha llegado a una hoja y el constructor `Rbin` con un entero y dos subárboles "levantados" (es decir, los que tienen el parámetro anidado).

```
data Ctree a = Pt a
             | Leaf
             | Rbin Int (Ctree (Lift a)) (Ctree (Lift a))

data Lift a = Zero | Suc a
```

El constructor `Pt` no representa un árbol, sino un apuntador hacia la raíz del árbol (`Zero`) o hacia otra posición (`Suc p`).

El tipo `Lift` maneja las posiciones de los apuntadores como numerales de la misma forma que las listas cíclicas: el apuntador de un nodo N que se dirige a la raíz es `Pt Zero`, un apuntador `Pt (Suc Zero)` de N se dirige al nodo `r1` que es el siguiente después de la raíz en la trayectoria de la raíz hacia N, un apuntador `Pt (Suc (Suc Zero))` de N se dirige al nodo `r2` que es el siguiente después de `r1` en la trayectoria antes mencionada, y así sucesivamente. Como ejemplo se da un árbol con dos ciclos (`arbol1`) que correesponde al de la figura 3.5.

```
arbol1 = Rbin 1
             (Rbin 2
                (Rbin 4
                    (Rbin 7 Leaf Leaf)
                    Leaf
                )
                (Rbin 5
                    (Rbin 8 (Pt (Suc Zero)) Leaf)
                    Leaf
                )
             )
             (Rbin 3
                 (Rbin 6
                    (Pt Zero)
                    (Rbin 9 Leaf Leaf)
                 )
                 Leaf
             )
```

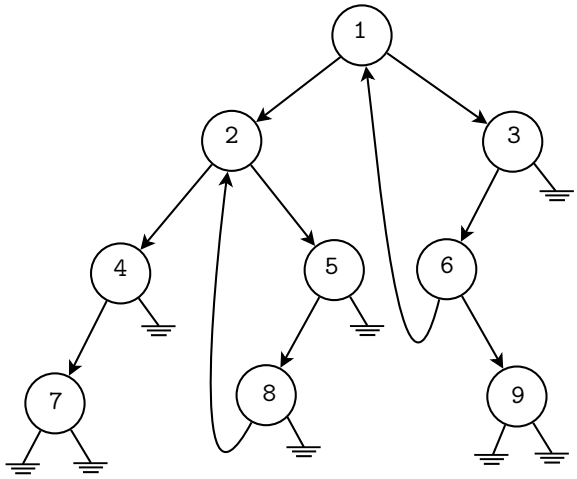


Figura 3.5: Árbol cíclico `arbol1`

Funciones constructoras

La función `cbin` genera un árbol binario cíclico a partir de un entero (raíz) y dos árboles cíclicos.

```
cbin :: Int -> Ctree a -> Ctree a -> Ctree a
cbin x xsL xsR = Rbin x (shiftT xsL) (shiftT xsR)
```

```
shiftT :: Ctree a -> Ctree (Lift a)
shiftT (Pt z) = Pt (Suc z)
shiftT Leaf = Leaf
shiftT (Rbin x xsL xsR) = Rbin x (shiftT xsL) (shiftT xsR)
```

La función `shift` funciona como levantador de árboles, en el sentido que al incorporarse un árbol como subárbol de otro, los ciclos deben ajustar sus apuntadores (análogo a lo que ocurre con listas cíclicas). Por ejemplo, si se aplica la función `cbin` a la raíz `x` (figura 3.6) y a dos árboles cíclicos `xsL` y `xsR` (figuras 3.7 y 3.8 respectivamente) devuelve un nuevo árbol `ntree` (figura 3.9).

Figura 3.6: Entero raíz `x`

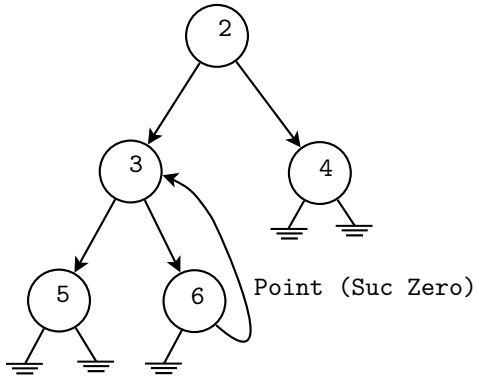


Figura 3.7: Subárbol izquierdo `xsL`

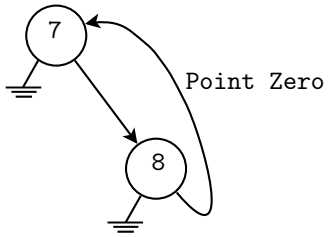


Figura 3.8: Subárbol derecho `xsR`

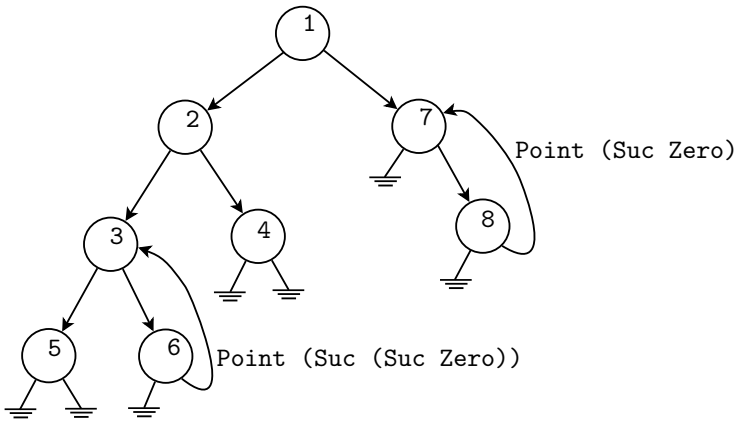


Figura 3.9: Nuevo árbol cíclico `ntree = cbin x xsL xsR`

En la figura 3.9 se puede observar que el ciclo correspondiente al nodo 6 en el árbol `xsL` cambió de `Point (Suc Zero)` a `Point (Suc (Suc Zero))` cuando se agregó como subárbol del árbol `ntree`. Del mismo modo el apuntador del nodo 8 en `xsR` cambió de `Point Zero` a `Point (Suc Zero)` en `ntree`.

Funciones destructoras

La función que obiene el nodo raíz de un árbol cíclico es similar a la de listas cíclicas. Pero es importante resaltar que un apuntador no es un árbol y que una hoja no tiene información.

```
croot :: Ctree a -> Int
croot (Pt z) = error "no es un árbol"
croot Leaf = error "hojas no tienen información"
croot (Rbin x _ _) = x
```

Para el caso de los árboles cíclicos la función que obtiene el subárbol izquierdo y derecho es diferente, porque los árboles son estructuras no lineales. Por ejemplo si queremos obtener al subárbol izquierdo de un árbol cíclico, el cual contiene un apuntador a la raíz, el resultado contiene no sólo una copia acomodada de la raíz sino tambien el subárbol derecho. Por ejemplo, consideremos el árbol cíclico de la figura 3.10, en el que el apuntador proveniente del nodo 3 va hacia la raíz. El subárbol izquierdo de `arbol2` se observa en la figura 3.11.En este ejemplo al destruir la estructura en el subárbol izquierdo, el derecho no se pierde, ya que el nodo 3 tiene un apuntador a la raíz. ¿Cómo guardar el estado del otro subárbol (en este caso el derecho) al hacer la llamada recursiva y además mantener la coherencia de los apuntadores? A continuación se ilustrará a detalle el uso de constructores apropiados (como los usados en listas cíclicas) para resolver este tipo de problemas.

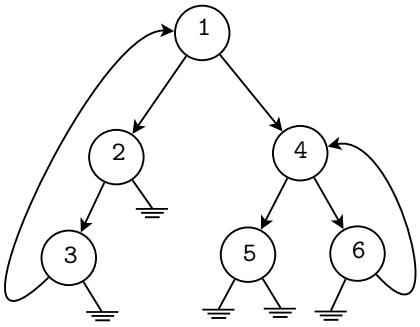


Figura 3.10: Árbol cíclico `arbol2`

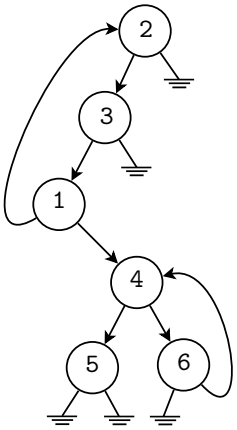


Figura 3.11: Subárbol izquierdo de `arbol2`

Se propone la función `csubL` para obtener el subárbol izquierdo de un árbol binario cíclico:

```
csubL :: Ctree a -> Ctree a
csubL (Pt z) = error "no es un árbol"
csubL Leaf = error "hoja no tiene subárboles"
csubL (Rbin x xsL xsR) = cnibL x xsL xsR

cnibL :: Int -> Ctree (Lift a) -> Ctree (Lift a) -> Ctree a
cnibL y (Pt Zero)        ys = Rbin y (Pt Zero) ys
cnibL y (Pt (Suc z))     ys = Pt z
cnibL y Leaf             ys = Leaf
cnibL y (Rbin x xsL xsR) ys = Rbin x (cnibL y xsL ys')
                                     (cnibL y xsR ys')
```

```
                                    where ys' = shiftT ys
```

No se puede obtener el subárbol izquierdo de una hoja porque no tiene subárboles, tampoco el de un apuntador porque al igual que sucede con las listas cíclicas el constructor `Pt` no representa un árbol cíclico.

Cuando `csubL` se aplica a un árbol binario llama a `cnibL`, que es una función recursiva que regresa el subárbol izquierdo (y su nombre se obtiene al invertir la palabra bin en `cbin`, ya que mientras `cbin` construye un árbol cíclico, `cnibL` lo destruye). Las variables `y` y `ys` son la raíz y el subárbol derecho respectivamente.

La función `cnibL` desplaza recursivamente la raíz de la lista junto con el subárbol derecho (ajustando mediante `shiftT` los apuntadores que en éste pudieran ocurrir) tantas veces hacia abajo hasta llegar a cada apuntador u hoja, en cuyo caso, la permanencia de la raíz y su subárbol derecho (ajustado) en esa posición se decide dependiendo de dicho apuntador u hoja, son tres casos:

1: `cnibL y (Pt Zero) ys = Rbin y (Pt Zero) ys`

Si es un apuntador a la raíz (`Pt Zero`), ella y el subárbol derecho se quedan es esa posición como ocurre en la figura 3.10 en donde el nodo tres tiene un apuntador a la raíz y al obtener el subárbol izquierdo en la figura 3.11 se observa que del nodo tres tiene a la raíz y al subárbol derecho.

2: `cnibL y (Pt (Suc z)) ys = Pt z`

Si es un apuntador a un nodo subsecuente (`Suc z`, en donde `z` es cualquier constructor de tipo `Lift`) se ajusta, decrementánndolo en uno porque el árbol perdió su raíz. La figura 3.13 muestra el subárbol izquierdo de `arbol3` (figura 3.12). En este caso se descarta la raíz y el subárbol derecho.

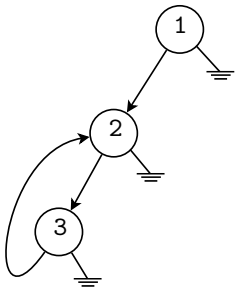


Figura 3.12: `arbol3`

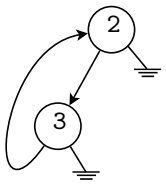


Figura 3.13: Subárbol izquierdo de `arbol3`

3: `cnibL y Leaf ys = Leaf`

Si es una hoja se deja igual, también se descarta la raíz y el subárbol derecho.

La función que obtiene el subárbol derecho es análoga:

```
csubR :: Ctree a -> Ctree a
csubR (Pt z) = error "no es un árbol"
csubR Leaf = error "hoja no tiene subárboles"
csubR (Rbin x xsL xsR) = cnibR x xsL xsR

cnibR :: Int -> Ctree (Lift a) -> Ctree (Lift a) -> Ctree a
cnibR y ys (Pt Zero) = Rbin y ys (Pt Zero)
cnibR y ys (Pt (Suc z)) = Pt z
cnibR y ys Leaf = Leaf
cnibR y ys (Rbin x xsL xsR) = Rbin x (cnibR y ys' xsL)
                                     (cnibR y ys' xsR)
                              where ys' = shiftT ys
```

A continuación se propone un forma desplegar árboles cíclicos como listas potencialmente infinitas.

Aplanado de árboles cíclicos

La función de aplanado (*flat*) del árbol cíclico representado por un árbol cíclico consiste en expresarlo como una lista común. Dado que el árbol es infinito, el recorrido que se utiliza para generar la lista es por anchura; ya que es mejor desplegar la información de arriba hacia abajo y así evitar el riesgo de caer en ciclos.

La estrategia utilizada para desplegar listas cíclicas (sección 3.2) consiste en utilizar directamente el operador `unfoldr` de listas regulares, debido al tipado de las funciones `cheadtail` y `unfoldr`:

```
cheadtail :: Clist a -> Maybe (Int,Clist a)

unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
```

sustituyendo `b` por `Clist a` en la signatura de `unfoldr` obtenemos la conversión de `Clist a` en `[a]`.

Sin embargo no es posible aplicar `unfoldr` a árboles cíclicos porque `Maybe` pide dos parámetros y al destruir árboles tenemos tres: raíz, subárbol izquierdo y derecho.

Una propuesta de solución a ese problema es definir primero un `unfoldr` para árboles generales (`Tree`), convertir `Ctree a` a `Tree` con ayuda de ese operador y después aplicar el recorrido por anchura al árbol resultante.

La implementación del tipo `Tree` de árboles regulares está predefinida en HASKELL como estructura de uso general[31]. La función que nos interesa es `unfoldTree` que construye un árbol `Tree` a partir de un valor inicial `b` (o semilla).

```
unfoldTree :: (b -> (Int, [b])) -> b -> Tree
```

De este modo se va a generar un árbol de tipo `Tree` a partir de una semilla que va a ser un árbol cíclico `Ctree a` (como se ha manejado en la función `cunfoldL` de listas cíclicas). El último paso para convertir un árbol cíclico en un árbol de tipo `Tree` es crear una función de tipo `b -> (Int, [b])` parecida a `Maybe`.

```
cizqder :: Ctree a -> (Int,[Ctree a])
cizqder Leaf = (-1,[ ])
cizqder xs = (croot xs,[csubL xs,csubR xs])
```

Esta función separa un árbol cíclico con los destructores `croot`, `csubL` y `csubR`, cabe mencionar que las hojas no tienen información y por simplicidad se les asigna el entero -1.

Una vez realizado el cambio de `Ctree a` a `Tree` se utiliza la función `levels` [5] que produce el recorrido por anchura de un árbol `Tree` (la lista de los elementos de cada nivel).

```
unwindTree :: Ctree a -> [[Int]]
unwindTree t = levels (unfoldTree cizqder t)
```

Por ejemplo para el árbol de la figura 3.12 el aplanado se visualiza de la siguiente manera:

```
> unwindTree arbol3
    [[1],[2,-1],[3,-1],[2,-1],[3,-1],[2,-1],[3,-1],...
```

[5] Definida también en [31] con tipo `levels :: Tree a ->[[a]]`.

La anterior es una lista de listas de enteros, cada una contiene a los elementos de cada nivel del árbol cíclico `arbol3`, las hojas se representan con `-1`.

Para obtener una lista de tipo `[a]` aplicamos a `unwindTree` la función `concat`, la cual transforma listas de listas en una sola lista. Con esto se define la función de aplanado de árboles cíclicos (transformando `Ctree a` en `[Int]`):

```
unwindTreeC :: Ctree a -> [Int]
unwindTreeC t = concat (levels (unfoldTree cizqder t))
```

```
> concat (unwindTreeC arbol3)
     [1,2,-1,3,-1,2,-1,3,-1,2,-1,3,-1,2,-1,3,...
```

unfold y map

A continuación se presenta el operador de desplegado `cunfoldTree` para árboles cíclicos, así como su aplicación inmediata para definir la función map. Estas funciones siguen el esquema de `cizqder` como función de opción.

```
cunfoldTree :: Eq c => (c -> (Int,[c])) -> c -> Ctree a
cunfoldTree = cunfoldT' [ ] [ ]

cunfoldT' :: Eq c => [c] -> [a] -> (c -> (Int,[c])) -> c
-> Ctree a
cunfoldT' lc la izqder c = case lookup c (zip lc la) of
  Nothing -> case izqder c of
    (-1,_) -> Leaf
    (x,[iz,de]) -> let lc' = lc ++ [c];
                       la' = Zero : map Suc la
                   in Rbin x (cunfoldT' lc' la' izqder iz)
                             (cunfoldT' lc' la' izqder de)
  Just a -> Pt a
```

Esta función es similar a la de listas cíclicas (`cunfoldL`) con la diferencia de que en este caso los estados que se guardan son listas que contienen los dos subárboles que obtiene la función de opción (`izqder`) y se debe aplicar la llamada recursiva dos veces (en el árbol izquierdo y en el derecho).

La función *map* para árboles cíclicos usando `cunfoldTree` es:

```
cmapTree :: Eq (Ctree a) => (Int -> Int) -> Ctree a -> Ctree b
cmapTree f xs = cunfoldTree izde xs where
  izde xs = case cizqder xs of
    (-1,[ ]) -> (-1,[ ])
    (x,xs') -> (f x,xs')
```

como ejemplo se utiliza también el árbol cíclico `arbol3` de la figura 3.12.

```
> show arbol3
     Rbin 1 (Rbin 2 (Rbin 3 (Pt (Suc Zero)) Leaf) Leaf) Leaf


> show (cmapTree (+10) arbol3)
     Rbin 11 (Rbin 12 (Rbin 13 (Pt (Suc Zero)) Leaf) Leaf) Leaf
```

Capítulo 4

Conclusiones

Hasta el momento en que se redacta este trabajo, la programación imperativa es más usada respecto a la funcional en la construcción de programas. La brecha ha disminuido gradualmente, sin embargo, para que los programadores utilicen herramientas de programación funcional en su rutina diaria es necesario poder escribir más y mejores estructuras de datos que puedan ser incorporadas en desarrollos de software de distinta naturaleza.

En este trabajo se implementó una pequeña biblioteca de estructuras cíclicas basada en el artículo de Neil Ghani, et al. *Representing Cyclic Structures as Nested Datatypes*[8] con el uso de tipos anidados, mostrando las ventajas de la programación funcional con Haskell como la evaluación perezosa (útil al manejar estructuras infinitas), la pureza, el tipado fuerte y el desarrollo de programas en pocas líneas de código. Además con estas estructuras se definieron funciones de orden superior como map y unfold.

Los tipos anidados no son utilizados frecuentemente debido a su desconocimiento y a la falta de implementaciones disponibles. Sin embargo los programas que los contienen se pueden verificar con herramientas formales como la teoría de tipos[1].

Un aspecto importante al definir las listas y árboles cíclicos con tipos anidados es que nos garantizan invariantes estructurales, es decir, cualquier lista definida con nuestro tipo `Clist` que pase la verificación de tipos del intérprete de Haskell es una lista cíclica (lo mismo ocurre con los árboles cíclicos) y así se evita definir y aplicar una función que verifique si una estructura (lista o árbol) es cíclica. Sin embargo esto tiene un costo (temporal), si bien acceder a los componentes de una lista me lleva tiempo constante en una estructura imperativa, puede llevar hasta tiempo cuadrático con las estructuras funcionales. Ese sacrificio de eficiencia viene unido al cuidado que el programador debe tener con el sistema de tipos en tareas como la recursión, la cual (como se mostró en el capítulo 3) no es como la de los tipos regulares.

Como posible rumbo de investigación después de este trabajo, se puede probar la generalización de la estructura anidada de árbol cíclico de modo que incluya *sharing*, es decir, que pueda haber apuntadores de un nodo a cualquiera de sus parientes, no necesariamente a uno que esté en el camino desde la raíz. Esta variante ya ha sido abordada en trabajos como el de Makoto Hamana[11] y el de Masahito Hasegawa[12], sin embargono utilizan tipos anidados. En mi opinión esto incrementa la complejidad, por ejemplo, al definir funciones básicas como la de obtener el subárbol derecho, si tiene un apuntador del nodo a (en el subárbol derecho) al nodo b (en el subárbol izquierdo) y a su vez b tiene un apuntador a la raíz, se puede guardar la raíz en el resultado y colocarla donde sea conveniente, pero no hay una forma trivial de guardar la referencia y la posición del nodo b en el resultado. Este es un tema que tiene aún mucho material por ofrecer, muchas estructuras funcionales pueden manejar datos eficientemente si sabemos las bases teóricas para definirlas.

Apéndice A

Árboles binarios completos

A.1. Implementación no anidada

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
            deriving(Show,Eq)

perfect :: Tree a -> Bool
perfect (Leaf x) = True
perfect (Node t1 t2) = perfect t1 && perfect t2
                                  && (hg t1 == hg t2)


-- Altura de un árbol
hg :: Tree a -> Int
hg (Leaf x) = 0
hg (Node t1 t2) = 1 + max (hg t1) (hg t2)


-- Aplanamiento
flat :: Tree a -> [a]
flat (Leaf x) = [x]
flat (Node t1 t2) = flat t1 ++ flat t2


-- map
mapt :: (a -> b) -> Tree a -> Tree b
mapt f (Leaf x) = Leaf (f x)
mapt f (Node t1 t2) = Node (mapt f t1) (mapt f t2)
```

```
-- Ejemplos
a1 = Node
       (Node (Leaf 1) (Leaf 2))
       (Leaf 3)

a2 = Node
       (Node
          (Node (Leaf 1) (Leaf 2))
          (Node (Leaf 3) (Leaf 4))
       )
       (Node
          (Node (Leaf 5) (Leaf 6))
          (Node (Leaf 7) (Leaf 8))
       )
```

A.2. Implementación anidada

```
data Perfect a = Zero a | Succ (Perfect (a,a))


-- Altura de un árbol
ap :: Perfect a -> Int
ap (Zero _) = 0
ap (Succ p) = (ap p) + 1


-- Aplanamiento

flF fla (x,y) = (fla x) ++ (fla y)

flp :: (a -> [x]) -> (Perfect a -> [x])
flp fla (Zero x) = fla x
flp fla (Succ pt) = flp (flF fla) pt

flatp = flp wrap
wrap x = [x]

-- map
mapt :: (a -> b) -> (Perfect a -> Perfect b)
mapt f (Zero x) = Zero (f x)
mapt f (Succ tp) = Succ (mapt (pairf f) tp)

pairf f (x,y) = (f x,f y)
```

Apéndice B

Estructuras cíclicas

B.1. Listas cíclicas anidadas

```
import Data.List

data Clist a = Point a
             | Rcons Int (Clist (Lift a))


data Lift a = Zero | Suc a


-- Constructor
ccons :: Int -> Clist a -> Clist a
ccons x xs = Rcons x (shift xs)

shift :: Clist a -> Clist (Lift a)
shift (Point z) = Point (Suc z)
shift (Rcons x xs) = Rcons x (shift xs)


-- Cabeza de una lista cíclica
chead :: Clist a -> Int
chead (Point z) = -1
chead (Rcons x _) = x


-- Cola de una lista cíclica

ctail :: Clist a -> Clist a
```

```
ctail (Point z) = error "caso tail"
ctail (Rcons x xs) = csnoc x xs

csnoc :: Int -> Clist (Lift a) -> Clist a
csnoc y (Point Zero) = Rcons y (Point Zero)
csnoc y (Point (Suc z)) = Point z
csnoc y (Rcons x xs) = Rcons x (csnoc y xs)


-- Desdoblar listas cíclicas

cheadtail :: Clist a -> Maybe (Int,Clist a)
cheadtail (Point z) = Nothing
cheadtail xs = Just (chead xs,ctail xs)

unwind = unfoldr cheadtail


-- Unfold de listas cíclicas

cunfoldL :: Eq c => (c -> Maybe (Int,c)) -> c -> Clist a
cunfoldL = cunfoldL' [] []

cunfoldL' :: Eq c => [c] -> [a] -> (c -> Maybe (Int,c))
                     -> c -> Clist a
cunfoldL' lc la ht c = case lookup c (zip lc la) of
  Nothing -> case ht c of
    Nothing -> error "nothing"
    Just (x,c') -> let lc' = lc ++ [c];
                       la' = Zero : map Suc la
                   in Rcons x (cunfoldL' lc' la' ht c')
  Just a -> Point a


-- map y zipWith para listas cíclicas

cmap f xs = cunfoldL ht xs where
    ht xs = case cheadtail xs of
        Nothing -> Nothing
        Just (x,xs') -> Just (f x,xs')


czipWith f xs ys = cunfoldL ht (xs,ys) where
    ht (xs,ys) = case cheadtail xs of
        Nothing -> Nothing
        Just (x,xs') -> case cheadtail ys of
```

```
            Nothing -> Nothing
            Just (y,ys') -> Just (f x y,(xs',ys'))

-- Ejemplos

clist1 = Rcons 1 (Rcons 2 (Point Zero))

clist2 = Rcons 1 (Rcons 2
(Rcons 3 (Point (Suc Zero))))

clist3 = Rcons 1 (Rcons 2 (Rcons 3 (Rcons 4
(Rcons 5 (Point (Suc (Suc (Zero))))))))
```

B.2. Árboles cíclicos anidados

```
import Data.Tree

-
Árboles binarios cíclicos de enteros sin
información en las hojas, por ejemplo para
mostrar un árbol arbol1 se escribe:

   > show arbol1
-
data Ctree a = Pt a
               | Leaf
               | Rbin Int (Ctree (Lift a)) (Ctree (Lift a))
               deriving(Show,Eq)

data Lift a = Zero | Suc a deriving(Show,Eq)


-- Constructor

cbin :: Int -> Ctree a -> Ctree a -> Ctree a
cbin x xsL xsR = Rbin x (shiftT xsL) (shiftT xsR)


shiftT :: Ctree a -> Ctree (Lift a)
shiftT (Pt z) = Pt (Suc z)
shiftT Leaf = Leaf
shiftT (Rbin x xsL xsR) = Rbin x (shiftT xsL) (shiftT xsR)
```

```
-- Raíz de un árbol cíclico
croot :: Ctree a -> Int
croot (Pt z) = error "no es un árbol"
croot Leaf = error "hojas no tienen información"
croot (Rbin x _ _) = x


-- Subárbol izquierdo de un árbol cíclico

csubL :: Ctree a -> Ctree a
csubL (Pt z) = error "no es un árbol"
csubL Leaf = error "hoja no tiene subárboles"
csubL (Rbin x xsL xsR) = cnibL x xsL xsR


cnibL :: Int -> Ctree (Lift a) -> Ctree (Lift a)
         -> Ctree a
cnibL y (Pt Zero)        ys = Rbin y (Pt Zero) ys
cnibL y (Pt (Suc z))     ys = Pt z
cnibL y Leaf             ys = Leaf
cnibL y (Rbin x xsL xsR) ys = Rbin x (cnibL y xsL ys')
                                     (cnibL y xsR ys')
                              where ys' = shiftT ys


-- Subárbol derecho de un árbol cíclico

csubR :: Ctree a -> Ctree a
csubR (Pt z) = error "no es un árbol"
csubR Leaf = error "hoja no tiene subárboles"
csubR (Rbin x xsL xsR) = cnibR x xsL xsR

cnibR :: Int -> Ctree (Lift a) -> Ctree (Lift a)
         -> Ctree a
cnibR y ys (Pt Zero) = Rbin y ys (Pt Zero)
cnibR y ys (Pt (Suc z)) = Pt z
cnibR y ys Leaf = Leaf
cnibR y ys (Rbin x xsL xsR) = Rbin x (cnibR y ys' xsL)
                                     (cnibR y ys' xsR)
                              where ys' = shiftT ys


-- Función de paso para unfoldr

cizqder :: Ctree a -> (Int,[Ctree a])
```

```
cizqder Leaf = (-1,[])
cizqder xs = (croot xs,[csubL xs,csubR xs])

-
Aplanado de árboles cíclicos por niveles.

Para prueba usar función take para evitar
el despliegue potencialmente infinito de
la lista por niveles:

    > take 50 unwindTree arbol3
-
unwindTree :: Ctree a -> [[Int]]
unwindTree t = levels (unfoldTree cizqder t)

-
Aplanado de árboles cíclicos

Para prueba:

    > take 30 unwindTreeC arbol3
-
unwindTreeC :: Ctree a -> [Int]
unwindTreeC t = concat (levels (unfoldTree cizqder t))


-- unfold general para árboles cíclicos

cunfoldTree :: Eq c => (c -> (Int,[c])) -> c -> Ctree a
cunfoldTree = cunfoldT' [] []

cunfoldT' :: Eq c => [c] -> [a] -> (c -> (Int,[c]))
                     -> c -> Ctree a
cunfoldT' lc la izqder c = case lookup c (zip lc la) of
  Nothing -> case izqder c of
    (-1,_) -> Leaf
    (x,[iz,de]) -> let lc' = lc ++ [c];
                       la' = Zero : map Suc la
                   in Rbin x (cunfoldT' lc' la' izqder iz)
                             (cunfoldT' lc' la' izqder de)
  Just a -> Pt a
```

```
-- map para árboles cíclicos
cmapTree :: Eq (Ctree a) => (Int -> Int) -> Ctree a -> Ctree a
cmapTree f xs = cunfoldTree izde xs where
  izde xs = case cizqder xs of
    (-1,[]) -> (-1,[])
    (x,xs') -> (f x,xs')


-- Ejemplos

arbol1 = Rbin 1
             (Rbin 2
                (Rbin 4
                    (Rbin 7 Leaf Leaf)
                    Leaf
                )
                (Rbin 5
                    (Rbin 8 (Pt (Suc Zero)) Leaf)
                    Leaf
                )
             )
             (Rbin 3
                 (Rbin 6
                    (Pt Zero)
                    (Rbin 9 Leaf Leaf)
                 )
                 Leaf
             )
```

```
arbol2 = Rbin 1
             (Rbin 2
                 (Rbin 3
                      (Pt Zero)
                      Leaf
                 )
                 Leaf
             )
             (Rbin 3
                 (Rbin 4
                      (Rbin 5
                         Leaf
                         Leaf
                      )
                      (Rbin 6
                         Leaf
                         Leaf
                      )
                 )
                 Leaf
              )

arbol3 = Rbin 1
            (Rbin 2
                (Rbin 3
                    (Pt (Suc Zero))
                    Leaf
                )
                Leaf
            )
     Leaf
```

Bibliografía

[1] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1):3-66,2005.

[2] Richard Bird and Lambert Meertens. Nested datatypes. *Mathematics of program construction*. Springer Berlin Heidelberg, 1998.

[3] Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming* , 9(1):77–91, 1999.

[4] G. Blelloch, H. Burch, K. Crary, R. Harper , G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming*, 11(5):441-466, 2001.

[5] Luca Cardelli. Type systems.*ACM Computing Surveys* 28(1):263-264. 1996.

[6] James Driscoll, Neil Sarnak, Daniel Sleator, and Robert Tarjan. Making data structures persistent. *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, p. 109-121. ACM. 1986.

[7] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space).*Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, p. 284-294. ACM. 1996.

[8] Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. *Proc. of 7th Symposium on Trends in Functional Programming*, p. 173-188. 2006.

[9] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, p. 273-279. ACM, 1998.

[10] Lourdes del Carmen González Huesca, *Coinducción: de la teoría de categorías a la programación funcional*, Tesis de licenciatura, UNAM, Facultad de Ciencias, 2007.

[11] Makoto Hamana. Initial algebra semantics for cyclic sharing structures. *Typed Lambda Calculi and Applications*, p. 127-141. Springer-Verlag. 2009.

[12] Masahito Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. Distinguished Dissertation Series. Springer-Verlag. 1999.

[13] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197-218, 2006.

[14] Zoltán Horváth, Tamás Kozsik, and Máté Tejfel. Proving Invariants of Functional Programs. *SPLST*. 2003.

[15] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359-411. 1989.

[16] Paul Hudak and Joseph Fasel. A gentle introduction to Haskell. *ACM Sigplan Notices*, 27(5):1-52, 1992.

[17] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007.

[18] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355-372. 1999.

[19] Garrin Kimmell, et al. Equational reasoning about programs with general recursion and call-by-value semantics. *Proceedings of the sixth workshop on Programming languages meets program verification*, p. 15-26. ACM. 2012.

[20] Ralph Matthes, and Martin Strecker. Verification of the redecoration algorithm for triangular matrices. *Types for Proofs and Programs*, p. 125-141. Springer Berlin Heidelberg. 2008.

[21] John Mitchell *Foundations for programming languages*. MIT press, 1996.

[22] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[23] Benjamin Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[24] Fethi Rabhi and Guy Lapalme. *Algorithms; A Functional Programming Approach*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] Ben Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633-649. 1989.

[26] Simon Thompson. *Haskell: the craft of functional programming*. Vol. 2. Addison- Welsey, 1999.

[27] Favio Ezequiel Miranda Perea. Notas de clase para el curso de Lenguajes de programación y sus paradigmas.
Disponible en:
`http://www.matematicas.unam.mx/favio/cursos.html`
Fecha de consulta:10 de marzo de 2014.

[28] *The Haskell Programming Language. Functional programming.*
Disponible en:
`http://www.haskell.org/haskellwiki/Functional_programming`
Fecha de consulta: 10 de marzo de 2014.

[29] Simon Marlow (ed). *Haskell 2010 language report.*
Disponible en:
`http://www.haskell.org/onlinereport/haskell2010`
Fecha de consulta: 10 de marzo de 2014.

[30] *Haskell. The base package*
Disponible en:
`http://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html`
Fecha de consulta: 10 de marzo de 2014.

[31] *Haskell. Data.Tree*
Disponible en:
`http://hackage.haskell.org/package/containers-0.5.4.0/docs/Data-Tree.html`
Fecha de consulta: 10 de marzo de 2014.

[32] *Haskell. The array package.*
Disponible en:
`http://hackage.haskell.org/package/array-0.5.0.0/docs/Data-Array.html`
Fecha de consulta: 10 de marzo de 2014.

[33] *Haskell. The Char type and associated operations.*
Disponible en:
`http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-Char.html`
Fecha de consulta: 10 de marzo de 2014.