1 Introducción

El propósito al escribir este tutorial no es enseñar programación, ni siquiera la programación funcional. Por el contrario, el objetivo es servir como un suplemento al informe de Haskell (the Haskell Report) [4] (en lo sucesivo, simplemente el Informe), que es, por otro lado, una densa exposición técnica. Nuestro objetivo es proporcionar una introducción "agradable" a Haskell para aquellos que tienen experiencia con al menos otro lenguaje, preferiblemente funcional (o "casi-funcional, como ML o Scheme). Si el lector desea aprender más sobre el estilo funcional, recomendamos la lectura del texto de Bird *Introduction to Functional Programming* [1], o el libro de Davie *An Introduction to Functional Programming Systems Using Haskell* [2]. Un estudio pormenorizado de los lenguajes para la programación funcional y sus técnicas, incluyendo algunos principios de diseño utilizados en Haskell, puede verse en [3].

El lenguaje Haskell ha evolucionado de forma significativa desde su nacimiento en 1987. Este tutorial está enfocado al Haskell 98. Las versiones anteriores han quedado obsoletas por lo que recomendamos a los usuarios de Haskell a que usen Haskell 98. Existen varias extensiones de Haskell 98 que han sido implementadas eficientemente. Estas no forman parte formalmente de Haskell 98 y no serán tratadas en este tutorial.

Nuestra estrategia general para introducir las características del lenguaje son: motivar la idea, definir algunos términos, dar ejemplos, y dar referencias precisas al Informe para más detalles. Sin embargo, sugerimos que el lector ignore absolutamente los detalles hasta completar la lectura de ésta introducción. Por otro lado, el Prelude estándar de Haskell (véase el Apéndice A del Informe) y las librerías estándarizadas (localizables en el Informe de librerías (Library Report)[5]) contiene numerosos ejemplos útiles de código Haskell; recomendamos una lectura cuidadosa de estos ejemplos una vez completada la lectura del tutorial. Con ello observará la sensación agradable del código real de Haskell, y también se acostumbrará al conjunto estándar de funciones y tipos predefinidos.

Finalmente, en http://haskel.org podemos encontrar información sobre el lenguaje Haskell y sus implementaciones.

[También hemos tomado la decisión de no exponer demasiado las reglas sintácticas al principio. Más bien, las introducimos de forma incremental según sean demandadas por los ejemplos, y entre corchetes, tal como éste párrafo. Esto contrasta con el orden de exposición del Informe, aunque éste contiene una fuente de detalles bien documentada (las referencias tales como "Informe-Sección 2.1" se refieren a las secciones del Informe).]

Haskell es un lenguaje de programación fuertemente tipado (en términos anglófilos, a *typeful* programming language; término atribuido a Luca Cardelli.) Los tipos son penetrantes (pervasive), y el principiante deberá acostumbrarse desde el comienzo a la potencia y complejidad del sistema de tipos de Haskell . Para aquellos lectores que conozcan únicamente lenguajes no tipados, como Perl, Tcl, o Scheme, esto puede ser una dificultad adicional; para aquellos familiarizados con Java, C, Modula, o incluso ML, el acomodo a nuestro lenguaje será más significativo, puesto que el sistema de tipos de Haskell es diferente y algo más rico. En cualquier caso, la programación fuertemente tipada forma parte de la experiencia de la programación en Haskell, y no puede ser evitada.

2 Valores, Tipos, y otras Golosinas

Puesto que Haskell es un lenguaje funcional puro, todos los cómputos vienen descritos a través de la evaluación de *expresiones* (términos sintácticos) para producir *valores* (entidades abstractas que son vistas como respuestas). Todo valor tiene asociado un *tipo*. (Intuitivamente, podemos pensar que los tipos son conjuntos de valores.) Ejemplos de expresiones son los valores atómicos tales como el entero `5`, o el carácter `'a'`, o la función `\x -> x+1`, y los valores estructurados como la lista `[1,2,3]` y el par `('b',4)`.

Ya que las expresiones denotan valores, las expresiones de tipo son términos sintácticos que denotan *tipos*. Ejemplos de expresiones de tipo son los tipos atómicos `Integer` (enteros con precisión ilimitada), `Char` (caracteres), `Integer->Integer` (funciones que aplican `Integer` sobre `Integer`), así como los tipos estructurados `[Integer]` (lista homogénea de enteros) y `(Char,Integer)` (par formado por un carácter y un entero).

Todos los valores de Haskell son de primera categoría ("first-class") ---pueden ser argumentos o resultados de funciones, o pueden ser ubicados en estructuras de datos, etc. Por otro lado, los tipos de Haskell *no* son de primera categoría. En cierto sentido, los tipos describen valores, y la asociación de un valor con su tipo se llama un tipificado (*typing*). Usando los ejemplos anteriores, podemos escribir "tipificaciones" como los siguientes:

```
    5  :: Integer
  'a' :: Char
  inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

El símbolo "`::`" puede leerse "tiene el tipo".

En Haskell las funciones se definen usualmente a través de una colección de *ecuaciones*. Por ejemplo, la función `inc` puede definirse por una única ecuación:

```
inc n          = n+1
```

Una ecuación es un ejemplo de *declaración*. Otra forma de declaración es la declaración de tipo de una función o *type signature declaration* (§4.4.1), con la cual podemos dar de forma explícita el tipo de una función; por ejemplo, el tipo de la función `inc`:

```
inc            :: Integer -> Integer
```

Veremos más detalles sobre definiciones de funciones en la Sección 3.

Por razones pedagógicas, cuando queramos indicar que una expresión e_1 se evalúa, o "reduce" a otra expresión o valor e_2, escribiremos:

e_1 => e_2

Por ejemplo:

`inc (inc 3)` => `5`

El sistema de tipificación estático de Haskell define formalmente la relación entre tipos y valores (§4.1.4). Esta tipificación estática asegura que un programa Haskell está bien tipificado (*type safe*); es decir, que el programador no puede evaluar expresiones con tipos erróneos. Por ejemplo, no podemos sumar dos caracteres, ya que la expresión `'a'+'b'` está mal tipificada. La ventaja principal del tipificación estática es bien conocida: todos los errores de tipificado son detectados durante la compilación. No todos los errores son debidos al sistema de tipos; una expresión tal como `1/0` es tipificable pero su evaluación provoca un error en tiempo de ejecución. No obstante, el sistema de tipos puede encontrar errores durante la compilación, lo que proporciona al programador una ayuda para razonar sobre los programas, y también permite al compilador generar un código más eficiente (por ejemplo, no se requiere ninguna información de tipo o pruebas durante la ejecución).

El sistema de tipos también asegura que los tipos que el usuario proporciona para las funciones son correctos. De hecho, el sistema de tipos de Haskell es lo suficientemente potente como para describir cualquier tipo de función (con algunas excepciones que veremos más tarde) en cuyos caso diremos que el sistema de tipos *infiere* tipos correctos. No obstante, son aconsejables las oportunas declaraciones de tipos para las funciones, como la proporcionada para la función `inc`, ya que el tipificado de funciones es una forma eficiente de documentar y ayudar al programador a detectar errores.

[El lector habrá notado que los identificadores que comienzan con mayúscula denotan tipos específicos, tales como `Integer` y `Char`, pero no los identificadores que denotan valores, como `inc`. Esto no es un convenio: es obligatorio debido a la sintaxis de Haskell. Además, todos los caracteres, mayúsculas y minúsculas, son significativos: `foo`, `fOo`, y `fOO` son identificadores distintos.]

2.1 Tipos Polimórficos

Haskell proporciona tipos *polimóficos* ---tipos que son cuantificados universalmente sobre todos los tipos. Tales tipos describen esencialmente familias de tipos. Por ejemplo, (para_todo `a`)`[a]` es la familia de las listas de tipo base `a`, para cualquier tipo `a`. Las listas de enteros (e.g. `[1,2,3]`), de caracteres (`['a','b','c']`), e incluso las listas de listas de interos, etc., son miembros de esta familia. (Nótese que `[2,'b']` *no* es un ejemplo válido, puesto que no existe un tipo que contenga tanto a `2` como a `'b'`.)

[Los identificadores tales como el anterior `a` se llaman *variables de tipo*, y se escriben en minúscula para distinguirlas de tipos específicos, como `Integer`. Además, ya que Haskell solo permite el cuantificador universal, no es necesario escribir el símbolo correspondiente a la cuantificación universal, y simplemente escribimos `[a]` como en el ejemplo anterior. En otras palabras, todas las variables de tipos son cuantificadas universalmente de forma implícita.]

Las listas constituyen una estructura de datos comunmente utilizada en lenguajes funcionales, y constituyen una buena herramienta para mostrar los principios del polimorfismo. En Haskell, la lista `[1,2,3]` es realmente una abreviatura de la lista `1:(2:(3:[]))`, donde `[]` denota la lista vacía y `:` es el operador infijo que añade su primer argumento en la cabeza del segundo argumento (una lista). (`:` y `[]` son, respectivamente, los operadores `cons` y `nil` del lenguaje Lisp) Ya que `:` es asociativo a la derecha, también podemos escribir simplemente `1:2:3:[]`.

Como ejemplo de función definida por el usuario y que opera sobre listas, consideremos el problema de contar el número de elementos de una lista:

```
length                  :: [a] -> Integer
length []               =  0
length (x:xs)           =  1 + length xs
```

Esta definición es auto-explicativa. Podemos leer las ecuaciones como sigue: "La longitud de la lista vacía es 0, y la longitud de una lista cuyo primer elemento es `x` y su resto es `xs` viene dada por 1 más la longitud de `xs`." (Nótese el convenio en el nombrado: `xs` es el plural de `x`, y `x:xs` debe leerse: "una `x` seguida de varias `x`).

Este ejemplo, además de intuitivo, enfatiza un aspecto importante de Haskell que debemos aclarar: la comparación de patrones (*pattern matching)*. Los miembros izquierdos de las ecuaciones contienen patrones tales como `[]` y `x:xs`. En una aplicación o llamada a la función, estos patrones son comparados con los argumentos de la llamada de forma intuitiva (`[]` solo "concuerda" (matches) o puede emparejarse con la lista vacia, y `x:xs` se podrá emparejar con una lista de al menos un elemento, instanciándose `x` a este primer elemento y `xs` al resto de la lista). Si la comparación tiene éxito, el miembro izquierdo es evaluado y devuelto como resultado de la aplicación. Si falla, se intenta la siguiente ecuación, y si todas fallan, el resultado es un error.

La definición de funciones a través de comparación de patrones es usual en Haskell, y el usuario deberá familiarizarse con los distintos tipos de patrones que se permiten; volveremos a esta cuestión en la Sección 4.

La función `length` es también un ejemplo de función polimórfica. Puede aplicarse a listas con elementos de cualquier tipo, por ejemplo `[Integer]`, `[Char]`, o `[[Integer]]`.

```
length [1,2,3]       => 3

length ['a','b','c'] => 3

length [[1],[2],[3]] => 3
```

He aquí dos funciones polimórficas muy útiles sobre listas, que usaremos más tarde. La función `head` devuelve el primer elemento de una lista, y la función `tail` devuelve la lista salvo el primero:

```
head                    :: [a] -> a
head (x:xs)             =  x

tail                    :: [a] -> [a]
tail (x:xs)             =  xs
```

Al contrario que `length`, estas funciones no estan definidas para todos los posibles valores de su argumento. Cuando las funciones son aplicadas a la lista vacía se produce un error en tiempo de ejecución.

Vemos que algunos tipos polimórficos son más generales que otros en el sentido de que el conjunto de valores que definen es más grande. Por ejemplo, el tipo `[a]` es más general que `[Char]`. En otras palabras: el tipo `[Char]` puede ser derivado del tipo `[a]` a través de una sustitución adecuada de `a`. Con respecto a este orden generalizado, el sistema de tipos de Haskell tiene dos propiedades importantes: en primer lugar, se garantiza que toda expresión bien tipificada tenga un único tipo principal (descrito después), y en segundo lugar, el tipo principal puede ser inferido automáticamente (§4.1.4). En comparación con un lenguaje con tipos monomórficos como C, el lector encontrará que el polimórfismo enriquece la expresividad, y que la inferencia de tipos reduce la cantidad de tipos usados por el programador.

El tipo principal de una expresión o función es el tipo más general que, intuitivamente, "contiene todos los ejemplares de la expresión." Por ejemplo, el tipo principal de `head` es `[a]->a`; los tipos `[b]->a`, `a->a`, o el propio `a` son demasiado generales, mientras que algo como `[Integer]->Integer` es demasiado concreto. La existencia de un único tipo principal es la característica esencial del *sistema de tipos de Hindley-Milner*, que es la base del sistema de tipos de Haskell, ML, Miranda, ("Miranda" es marca registrada de Research Software, Ltd.) y otros lenguajes (principalmente funcionales) .

2.2 Tipos definidos por el usuario

Podemos definir nuestros propios tipos en Haskell a través de una declaración `data`, que introduciremos con una serie de ejemplos (§4.2.1).

Un dato predefinido importante en Haskell corresponde a los valores de verdad:

```
data Bool               = False | True
```

El tipo definido con tal declaración es `Bool`, y tiene exactamente dos valores: `True` y `False`. `Bool` es un ejemplo de *constructor de tipo* (sin argumentos), mientras que `True` y `False` son *constructores de datos* (o *constructores*, para abreviar).

En forma similar, podemos definir un tipo color:

```
data Color              = Red | Green | Blue | Indigo | Violet
```

Tanto `Bool` como `Color` son ejemplos de tipos enumerados, puesto que constan de un número finito de constructores.

El siguiente es un ejemplo de tipo con un solo constructor de dato:

```
data Point a            = Pt a a
```

Al tener un solo constructor, un tipo como `Point` es llamado a menudo un *tipo tupla*, ya que esencialmente es un producto cartesiano (en este caso binario) de otros tipos. (La tuplas son conocidas en otros lenguajes como registros.) Por el contrario, un tipo multi-constructor, tal como `Bool` o `Color`, se llama una "suma de tipos" o tipo unión (disjunta).

Sin embargo, lo más importante es que `Point` es un ejemplo de tipo polimórfico: para cualquier tipo t, define el tipo de los puntos cartesianos que usan t como eje de coordenadas. El tipo `Point` puede también verse como un constructor de tipos unario, ya

que a partir de un tipo t podemos obtener un nuevo tipo `Point t`. (En el mismo sentido, usando el ejemplo de la listas, `[]` es también un constructor de tipos: podemos aplicar el constructor `[]` a un tipo t para obtener un nuevo tipo `[t]`. La sintaxis de Haskell permite escribir `[t]` en lugar de `[] t`. Similarmente, `->` es otro constructor de tipos binario: dados dos tipos "t" y "u", `t->u` es el tipo de las funciones que aplican datos de tipo "t" a elementos de tipo "u".)

Nótese que el tipo del constructor de datos `Pt` es `a -> a -> Point a`, y las siguientes asignaciones de tipos son válidas:

```
Pt  2.0  3.0          :: Point Float
Pt  'a'  'b'          :: Point Char
Pt True False         :: Point Bool
```

Por otro lado, una expresión tal como `Pt 'a' 1` está erróneamente tipificada, ya que `'a'` y `1` son de tipos diferentes.

Es importante distinguir entre la aplicación de un *constructor de datos* para obtener un valor, y la aplicación de un *constructor de tipos* para obtener un *tipo*; el primero tiene lugar durante el tiempo de ejecución, que es cuando se computan cosas en Haskell, mientras que el último tiene lugar en tiempo de compilación y forma parte del proceso de tipificado que asegura un "tipo seguro".

[Constructores de tipo como `Point` y constructores de datos como `Pt` aparecen en niveles distintos de la declaración, lo que permite que el mismo nombre pueda usarse como constructor de tipos y como constructor de datos, como vemos en:

```
data Point a = Point a a
```

Esto puede llevar a una pequeña confusión al principio, pero sirve para crear un enlace obvio entre el constructor de datos y el de tipo.]

2.2.1 Tipos recursivos

Los tipos pueden ser recursivos, como el siguiente tipo para árboles binarios:

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

Con ello hemos definido un tipo polimórfico cuyos elementos son o bien hojas conteniendo un valor de tipo `a`, o nodos internos ("ramas") conteniendo (en forma recursiva) dos subárboles.

En la lectura de declaraciones de datos como la anterior, recordemos que `Tree` es un constructor de tipos, mientras que `Branch` y `Leaf` son constructores de datos. La declaración anterior, además de establecer una conexión entre estos constructores, define esencialmente los tipos para los constructores `Branch` y `Leaf`:

```
Branch                  :: Tree a -> Tree a -> Tree a
Leaf                    :: a -> Tree a
```

Con este ejemplo tenemos un tipo suficientemente rico que permite definir algunas funciones (recursivas) interesantes que hagan uso de éste. Por ejemplo, supongamos que queremos definir una función `fringe` que devuelva todos los elementos de las hojas de un árbol de izquierda a derecha. En primer lugar es esencial escribir el tipo de la nueva función; en este caso vemos que el tipo debe ser `Tree a -> [a]`. Es decir, `fringe` es una función polimórfica que, para cualquier tipo `a`, aplica árboles de `a` sobre listas de `a`. Una definición adecuada es la siguiente:

```
fringe                      :: Tree a -> [a]
fringe (Leaf x)             =  [x]
fringe (Branch left right) =  fringe left ++ fringe right
```

donde ++ es el operador infijo que concatena dos listas (su definición completa se verá en la Section 9.1). Al igual que la función `length` vista anteriormente, la función `fringe` está definida mediante comparación de patrones, salvo que los patrones implicados son los constructores de la definición dada por el usuario: `Leaf` y `Branch`. [Nótese que los parámetros formales son fácilmente identificados ya que comienzan con letras minúsculas.]

2.3 Sinónimos de Tipos

Por conveniencia, Haskell proporciona una forma para definir *sinónimos de tipos*; es decir, nombres de tipos usados varias veces. Los sinónimos de tipo son creados a través de una declaración `type` (§4.2.2). He aquí algunos ejemplos:

```
type String             = [Char]
type Person             = (Name,Address)
type Name               = String
data Address            = None | Addr String
```

Los sinónimos no definen tipos nuevos, sino simplemente proporcionan nuevos nombres a tipos ya existentes. Por ejemplo, el tipo `Person -> Name` es precisamente equivalente al tipo `(String,Address) -> String`. Los nombres nuevos son a menudo más cortos que los tipos nombrados, pero éste no es el único propósito de los sinónimos de tipos: éstos pueden mejorar la legibilidad de los programas a través de nemotécnicos; en efecto, los ejemplos anteriores enfatizan este hecho. Podemos dar nuevos nombres a tipos polimórficos:

```
type AssocList a b              = [(a,b)]
```

Este es el tipo de las "listas de asociaciones" que asocian valores de tipo `a` con otros de tipo `b`.

2.4 Los tipos predefinidos no son especiales

Antes hemos introducido varios tipos "predefinidos" tales como listas, tuplas, enteros y caracteres. También mostramos como el programador puede definir nuevos tipos. Además de una sintaxis especial ¿los tipos predefinidos tienen algo más de especial? La respuesta es *no*. La sintaxis especial es por conveniencia y consistencia, junto a algunas razones históricas, pero no tiene ninguna consecuencia semántica.

Enfatizamos este punto diciendo que la apariencia de las declaraciones de éstos tipos predefinidos es especial. Por ejemplo, el tipo `Char` puede ser descrito en la forma:

```
data Char       = 'a' | 'b' | 'c' | ...         -- Esto no es código
Haskell
                | 'A' | 'B' | 'C' | ...         -- válido!
                | '1' | '2' | '3' | ...
                ...
```

Los nombres de los constructores no son válidos sintácticamente; ello lo podríamos arreglar escribiendo algo como lo siguiente:

```
data Char       = Ca | Cb | Cc | ...
                | CA | CB | CC | ...
                | C1 | C2 | C3 | ...
                ...
```

Tales constructores son más concisos, pero no son los habituales para representar caracteres.

En cualquier caso, la escritura de código "pseudo-Haskell" tal como la anterior ayuda a aclarar la sintaxis especial. Vemos que `Char` es, en efecto, un tipo enumerado compuesto de un gran número de constructores (constantes). Por ejemplo, visto `Char` de esta forma aclaramos qué patrones pueden aparecer en las definiciones de funciones; es decir, qué constructores de este tipo podemos encontrarnos.

Este ejemplo también muestra el uso de los *comentarios* en Haskell; los caracteres `--` y los sucesivos hasta el final de la línea son ignorados. Haskell también permite comentarios *anidados* que tienen las forma `{-...-}` y pueden aparecer en cualquier lugar (§2.2).]

Similarmente, podemos definir `Int` (enteros de precisión limitada) y `Integer` en la forma:

```
data Int  = -65532 | ... | -1 | 0 | 1 | ... | 65532  -- más pseudo-código
data Integer =   ... -2 | -1 | 0 | 1 | 2 ...
```

donde `-65532` y `65532`, representan el mayor y el menor entero en precisión fija para una implementación concreta. `Int` es un tipo enumerado más largo que `Char`, pero es finito! Por el contrario, el pseudo-código para `Integer` (el tipo de los enteros con precisión arbitraria) debe verse como un tipo enumerado *infinito*.

Las tuplas también son fáciles de definir en la misma forma:

```
data (a,b)              = (a,b)                         -- más     peudo-
código
data (a,b,c)            = (a,b,c)
data (a,b,c,d)          = (a,b,c,d)
 .                        .
 .                        .
 .                        .
```

Cada una de las declaraciones anteriores define una tupla de una longitud particular, donde `(...)` juega distintos papeles: a la izquierda como constructor de tipo, y a la derecha como constructor de dato. Los puntos verticales después de la última declaración indican un

número infinito de tales declaraciones, reflejando el hecho de que en Haskell están permitidas las tuplas de cualquier longitud.

La listas son manipulables fácilmente, y lo que es más importante, son recursivas:

```
data [a]              = [] | a : [a]                  -- más peudo-código
```

Vemos que esto se ajusta a lo ya dicho sobre listas: `[]` es la lista vacía, y `:` es el constructor infijo de listas; de esta forma `[1,2,3]` es equivalente a la lista `1:2:3:[]`. (`:` es asociativo a la derecha.) El tipo de `[]` es `[a]`, y el tipo de `:` es `a->[a]->[a]`.

[De esta forma ":" está definido con una sintaxis legal---los constructores infijos se permiten en declaraciones `data`, y (para describir la comparación de patrones) son distinguidos de los operadores infijos ya que comienzan con el carácter ":" (una propiedad satisfecha trivialmente por ":").]

En este punto, el lector deberá notar con cuidado las diferencias entre tuplas y listas, ya que las definiciones anteriores lo aclaran suficientemente. En particular, nótese la naturaleza recursiva de las listas, con longitud arbitraria y cuyos elementos son homogéneos, y la naturaleza no recursiva de una tupla concreta, que tiene una longitud fija, en la cual los elementos son heterogéneos. Las reglas de tipificado para tuplas y listas deberían quedar claras ahora:

Para $(e_1, e_2, ..., e_n)$, n>=2, si t_i es el tipo de e_i, entonces el tipo de la tupla es $(t_1, t_2, ..., t_n)$.

Para $[e_1, e_2, ..., e_n]$, n>=0, cada e_i debe tener el mismo tipo t, y el tipo de la lista es `[t]`.

2.4.1 Listas por comprensión y Secuencias Aritméticas

Como en algunos dialectos de Lisp, las listas son muy útiles en Haskell, y al igual que en otros lenguajes funcionales, existe aún una sintaxis más adecuada para su descripción. Además de los constructores de listas ya introducidos, Haskell proporciona expresiones conocidas como *listas por comprensión* que introducimos con un ejemplo:

```
[ f x | x <- xs ]
```

Intuitivamente, esta expresión puede leerse como "la lista de todos los `f x` tales que `x` recorre `xs`." La similitud con la notación de los conjuntos no es una coincidencia. La frase `x <- xs` se llama un *generador*, y pueden utilizarse varios, como en:

```
[ (x,y) | x <- xs, y <- ys ]
```

Tal lista por comprensión determina el producto cartesiano de dos listas `xs` y `ys`. Los elementos son seleccionados como si los generadores fueran anidados de izquierda a derecha (con el de más a la derecha variando el último); es decir, si `xs` es `[1,2]` e `ys` es `[3,4]`, el resultado es `[(1,3),(1,4),(2,3),(2,4)]`.

Además de los generadores, se permiten expresiones booleanas llamadas *guardas* que establecen restricciones sobre los elementos generados. Por ejemplo, he aquí una definición compacta del algoritmo de ordenación favorito de todo el mundo:

```
quicksort  []           =  []
quicksort (x:xs)        =  quicksort [y | y <- xs, y<x ]
                        ++ [x]
                        ++ quicksort [y | y <- xs, y>=x]
```

Como otra ayuda en la descripción de listas, Haskell admite una sintaxis especial para *secuencias aritméticas*, que mostramos con una serie de ejemplos:

```
[1..10]   => [1,2,3,4,5,6,7,8,9,10]

[1,3..10] => [1,3,5,7,9]

[1,3..]   => [1,3,5,7,9, ...] (secuencia infinita)
```

Veremos algo más sobre secuencias aritméticas en la Sección 8.2, y sobre "listas infinitas" en la Sección 3.4.

2.4.2 Cadenas

Como otro ejemplo de sintaxis especial para tipos predefinidos, hacemos notar que la cadena de caracteres `"hello"` es una forma simplificada de la lista de caracteress `['h','e','l','l','o']`. Además, el tipo de `"hello"` es `String`, donde `String` es un sinónimo de tipo predefinido:

```
type String             = [Char]
```

Esto significa que podemos usar las funciones polimórficas sobre listas para operar con cadenas (strings). Por ejemplo:

```
"hello" ++ " world" => "hello world"
```

3 Funciones

Ya que Haskell es un lenguaje funcional, podemos esperar que las funciones jueguen un papel esencial, como veremos seguidamente. Es esta sección, trataremos varios aspectos de las funciones en Haskell.

En primer lugar, consideremos la siguiente defínción de función que suma sus dos argumentos:

```
add                     :: Integer -> Integer -> Integer
add x y                 =  x + y
```

Este es un ejemplo de función parcializada (*curried)*. (El origen del nombre *curry* proviene de la persona que popularizó el uso de la parcialización: Haskell Curry. Para capturar la función anterior en forma no parcializada (*uncurried*), podemos usar una *tupla*, como en:

```
add (x,y)               = x + y
```

¡Pero entonces tal versión de `add` es realmente una función de un solo argumento! Una aplicación de `add` tiene la forma `add` e_1 e_2, y es equivalente a `(add` e_1`)` e_2, ya que la aplicación de funciones es asociativa a la *izquierda*. En otras palabras, al aplicar `add` a un argumento se obtiene una nueva función que puede ser aplicada al segundo argumento. Esto es consistente con el tipo de `add`, `Integer->Integer->Integer`, ya que éste es equivalente a `Integer->(Integer->Integer)`; es decir, `->` es asociativo a la *derecha*. Además, usando `add`, podemos definir `inc` de forma distinta a la anterior:

```
inc                     = add 1
```

Este es un ejemplo de *aplicación parcial* de una función parcializada, y constituye un ejemplo de cómo podemos devolver una función como un valor. Consideremos ahora el caso en el que pasamos una función como argumento. La bien conocida función `map` constituye un ejemplo perfecto:

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

[La aplicación tiene mayor precedencia que cualquier operator infijo, y por tanto la segunda ecuación debe entenderse en la forma `(f x) : (map f xs)`.] La función `map` es polimórfica y su tipo indica claramente que su primer argumento es una función; nótese que las dos `a`'s deben ser instanciadas al mismo tipo (e igualmente para las `b`'s). Como ejemplo de uso de `map`, podemos incrementar los elementos de una lista:

```
map (add 1) [1,2,3] => [2,3,4]
```

Estos ejemplos muestran la naturaleza de primera-categoría que tienen las funciones, que se llaman funciones de *orden superior* al ser usadas de la forma anterior.

3.1 Abstraciones lambda

En lugar de usar ecuaciones para definir funciones, podemos definirlas en forma "anónima" vía una *lambda abstración*. Por ejemplo, podemos escribir una función

equivalente a `inc` en la forma `\x -> x+1`. Del mismo modo, la función `add` es equivalente a `\x -> \y -> x+y`. Las lambda abstracciones anidadas pueden escribirse en la siguiente otra forma equivalente `\x y -> x+y`. De hecho, las ecuaciones:

```
inc x                   = x+1
add x y                 = x+y
```

son formas abreviadas de:

```
inc                     = \x   -> x+1
add                     = \x y -> x+y
```

Más tarde veremos algo más sobre tales equivalencias.

En general, si `x` tiene por tipo t_1 y `exp` t_2, entonces `\x->exp` tiene por tipo t_1`->`t_2.

3.2 Operadores Infijos

Los operadores infijos son también funciones, y pueden definirse a través de ecuaciones. Por ejemplo, tenemos la siguiente definición para el operador de concatenación:

```
(++)                    :: [a] -> [a] -> [a]
[]     ++ ys            =  ys
(x:xs) ++ ys            =  x : (xs++ys)
```

[Desde el punto de vista léxico, los operadores infijos consisten únicamente de "símbolos", al contrario que los identificadores normales que son alfanuméricos (§2.4). Haskell no permite operadores prefijos, salvo el operador menos (`-`), que simultánemaente es infijo y prefijo.]

Otro ejemplo de operator infijo importante es la *composición de funciones*:

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

3.2.1 Secciones

Puesto que los operadores infijos son funciones, tiene sentido parcializarlas. En Haskell, la aplicación parcial de un operador infijo se llama una *sección*. Por ejemplo:

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

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

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

[Los paréntesis son obligatorios.]

La última forma de sección devuelve la función equivalente al operador infijo, y permite pasar un operador infijo como argumento de una función, como por ejemplo en `map (+) [1,2,3]` (¡el lector deberá verificar que la última expresión devuelve una lista de

funciones!). También es necesario el uso de los paréntesis al dar el tipo de los operadores infijos, como vimos en los ejemplos correspondientes a `(++)` y `(.)`.

Ahora vemos que `add` es precisamente `(+)`, y `inc` es `(+1)`. Por ello, las siguientes definiciones son correctas:

```
inc                     = (+ 1)
add                     = (+)
```

Podemos asociar un operador infijo a un valor funcional, ¿y al revés? Sí --- para ello simplemente escribimos su identificador entre apóstrofes. Por ejemplo, `x `add` y` es lo mismo que `add x y`. (Nótese que `add` es orlado de apóstrofes, no *comillas simples* como las usadas para denotar caracteres; es decir, `'f'` es un carácter, mientras que `` `f` `` es un operador infijo. Afortunadamente, la mayoría de los terminales ASCII distinguen las fuentes usadas en este manuscrito.) Algunas funciones se leen mejor de esta forma. Un ejemplo es el predicado predefinido `elem` que comprueba la pertenencia; la expresión `x `elem` xs` puede leerse como "`x` es un elemento de `xs`."

[Existen otras reglas sintácticas especiales para secciones que afectan a los operadores -(§3.5,§3.4).]

En este punto, el lector puede estar sorprendido sobre la cantidad de posibilidades para definir funciones. La decisión de proporcionar tantos mecanismo es histórica, y refleja parcialmente este deseo por consistencia (por ejemplo, infixidad versus funciones ordinarias).

3.2.2 Declaraciones de infixidad

Para cualquier operador o constructor podemos dar una *declaración de infixidad* (incluidos aquellos que corresponden a identificadores literales, como `` `elem` ``). Esta declaración especifica un nivel de precedencia de 0 to 9 (0 el menor; la aplicación tiene el mayor nivel de precedencia: 10), y una indicación de su asociatividad. Por ejemplo, las declaraciones de infixidad para `++` y `.` son:

```
infixr 5 ++
infixr 9 .
```

Ambas especifican asociatividad a la derecha, la primera con un nivel de precedencia 5, y 9 para la otra. La asociatividad a la izquierda se especifica con `infixl`, y la no-asociatividad con `infix`. También podemos declarar varios operadores con la misma declaración de infixidad. Si no se especifica la infixidad de un operador particular, ésta se toma por defecto como `infixl 9`. (Ver en §4.4.2 una definición detallada de las reglas de asociatividad.)

3.3 Las Funciones son no estrictas

Sea `bot` definida en la forma:

```
bot                     = bot
```

Es decir, `bot` es una expresión que no termina. En sentido abstracto, podemos denotar con

_|_ (leido "bottom") el *valor* de una expresión que no termina. Las expresiones que devuelven algún tipo de error en ejecución, tal como `1/0`, también tendrán este valor. Tal error es no recuperable: los programas no pueden continuar si se produce este error. Otros errores detectados por el sistema de E/S, tal como un error producido por "fin-de-fichero", son recuperables y son tratados de forma distinta. (Tal error de E/S no es realmente un error, sino una "excepción". Pueden verse más detalles sobre excepciones en la Sección 7.)

Una función `f` se dice *estricta* si, al ser aplicada a una expresión sin terminación, tampoco termina. En otras palabras, `f` es estricta si el valor de `f bot` is _|_. En la mayoría de los lenguajes de programación *todas* las funciones son estrictas. Pero este no es el caso de Haskell. Como ejemplo simple, consideremos `const1`, la función constante igual a 1, definida por:

```
const1 x                = 1
```

En Haskell, el valor de `const1 bot` es `1`. Operacionalmente hablando, ya que `const1` no "necesita" el valor de su argumento, el sistema nunca intentará evaluarlo, y no se produce un cómputo sin terminación. Por esta razón, las funciones no estrictas se llaman "perezosas", y sus argumentos son evaluados en forma "perezosa" o por necesidad.

Puesto que un error y una expresión que no termina son semánticamente iguales en Haskell, el razonamiento anterior también sirve para los errores. Por ejemplo, `const1 (1/0)` también será evaluada a `1`.

Las funciones no estrictas son muy poderosas en varias aplicaciones. La principal ventaja es que liberan al programador de algunas consideraciones sobre el orden de evaluación. Podemos pasar valores que contienen gran cantidad de cómputo como argumentos sin preocuparnos de los cómputos que necesitan realizarse. Un ejemplo importante son las estructuras de datos *infinitas*.

Otra forma de describir las funciones no estrictas es el hecho de que los cómputos de Haskell se describen a través de definiciones, al contrario que las *asignaciones* presentes en los lenguajes tradicionales. Leeremos la siguiente declaración

```
v                       = 1/0
```

como "definimos `v` como `1/0`" en lugar de "computar `1/0` y memorizar el resultado en `v`". La división por cero dará un error únicamente en el caso de que el valor de `v` sea necesario. Por la misma razón, tal declaración no implica cómputos. Las asignaciones se realizan en forma ordenada: el significado depende del orden en que se realicen. Sin embargo, las definiciones pueden presentarse en cualquier orden sin cambiar el significado.

3.4 Estructuras de datos "Infinitas"

Una ventaja de la naturaleza no estricta de Haskell es que los constructores de datos tampoco son estrictos. Esto no deberá sorprendernos ya que los constructores constituyen una clase especial de funciones (la diferencia es que pueden utilizarse en la comparación de patrones). Por ejemplo, el constructor de listas `(:)` no es estricto.

Los constructores no-estrictos permiten definir (conceptualmente) estructuras de datos *infinitas*. Presentamos alguna de ellas:

```
ones                    = 1 : ones
```

Más interesante es la función `numsFrom`:

```
numsFrom n              = n : numsFrom (n+1)
```

de forma que `numsFrom n` es la lista infinita de los enteros a partir de `n`. Con ésta podemos construir una lista infinita de cuadrados:

```
squares                 = map (^2) (numsfrom 0)
```

(nótese el uso de una sección de `^`, el operador infijo exponenciación.)

Naturalmente, eventualmente podemos pensar en extraer un trozo finito de la lista, y para ello existen una colección de funciones predefinidas en Haskell: `take`, `takeWhile`, `filter`, y otras. La definición de Haskell incluye un conjunto amplio de funciones y tipos predefinidos -- éste es llamado "Standard Prelude". El Prelude completo aparece en el Appendix A del Informe; ver el trozo denominado `PreludeList` con objeto de inspeccionar algunas funciones útiles que trabajan sobre listas. Por ejemplo, `take` extrae los primeros elementos de una lista:

`take 5 squares` => `[0,1,4,9,16]`

La definición anterior de `ones` es una *lista circular*. En la mayoría de los casos la perezosidad tiene un impacto importante en la eficiencia, ya que una implementación deberá representar tales listas como una estructura circular, con objeto de economizar espacio de memoria.

Otro ejemplo de uso de circularidad es la secuencia infinita de la sucesión de Fibonacci computada eficientemente a través de:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

donde `zip` es una función del Standard Prelude que devuelve una mezcla uno a uno de dos listas:

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

Nótese que `fib`, una lista infinita, aparece definida en términos de ella misma, como si se "mordiera la cola." Por ello, podemos dibujar su cómputo como se muestra en la Figura 1.

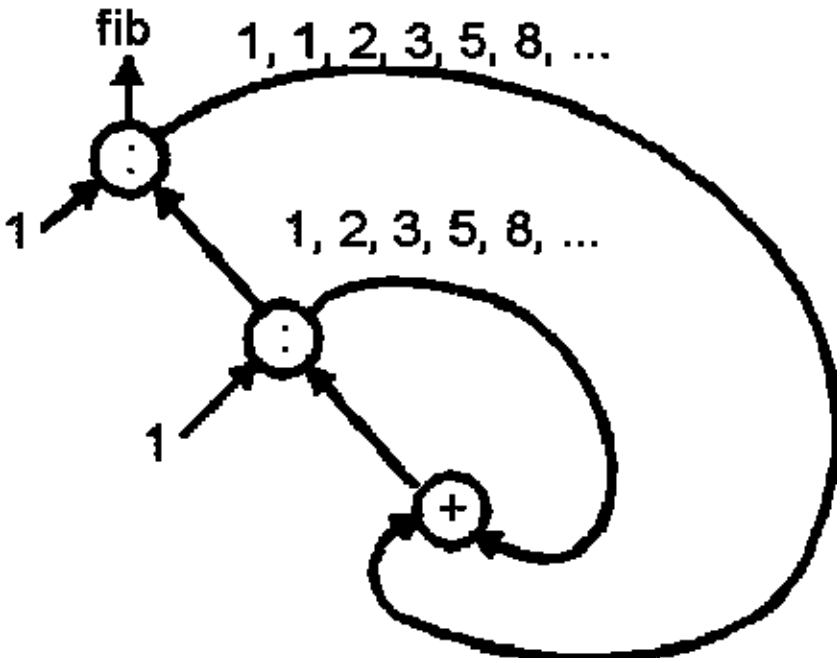

Figura 1

Otra aplicaciones de listas infinitas pueden verse en la Sección 4.4.

3.5 La función Error

Haskell dispone de una función predefinida llamada `error` y cuyo tipo es `String->a`. Es una función singular: de su tipo vemos que devuelve un valor de un tipo polimórfico sin ninguna información adicional, ya que no tiene un argumento que decida el tipo del resultado.

De hecho, *existe* un valor "compartido" por todos los tipos: _|_. Por supuesto, esto significa semánticamente que éste es el valor devuelto por `error` (recordemos que todos los errores tienen el mismo valor _|_).

Además, cualquier implementación razonable debería imprimir la cadena de caracteres argumento de `error` con objetivos informativos. Esta función es útil si queremos terminar un programa cuando algo "fue mal". Por ejemplo, la definición de `head` tomada del Standard Prelude es:

```
head (x:xs)             =  x
head  []                =  error "head{PreludeList}: head []"
```

4 Expresiones por Casos y Comparación de Patrones

Antes vimos varios ejemplos de comparación de patrones en definiciones de funciones- por ejemplo `length` y `fringe`. En esta sección revisaremos con detalle el proceso de comparación de patrones (§3.17). (La comparación de patrones de Haskell es diferente a la utilizada en los lenguajes de programación lógicos, como Prolog; en particular, en Haskell puede verse como una comparación "uni-direccional", mientras que en Prolog es "bi-direccional" (via la unificación), con vuelta-atrás implícita en el mecanismo de evaluación.)

Los patrones no son de "primera-categoría;" existe un conjunto fijo de patrones diferentes. Vimos varios ejemplos de patrones como los *constructores de datos*; las funciones `length` y `fringe` anteriores usan tales patrones, la primera usa constructores predefinidos (listas), y la segunda patrones definidos por el usuario (`Tree`). Por tanto, al comparar patrones se permiten constructores de cualquier tipo, definidos por el usuario o no. Esto incluye tuplas, cadenas, números, caracteres, etc. Por ejemplo, he aquí una función que concuerda tuplas de "constantes":

```
contrived :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrived    ([],  'b',  (1,   2.0),   "hi",   True) = False
```

Este ejemplo ilustra que se permite el anidamiento de patrones (de profundidades arbitrarias).

Técnicamente hablando, los *parámetros formales* (el Informe los llama *variables*.) son también patrones---para éstos *nunca falla* la comparación de patrones. Como "efecto lateral" de una comparación con éxito, la variable se instancia al valor comparado. Por esta razón, no se permite que una ecuación tenga variables repetidas como patrones. (propiedad que llamamos *linealidad* §3.17, §3.3,§4.4.3).

Patrones como las variables, que nunca fallan, se llaman *irrefutables*, en contraste con los patrones *refutable*s que pueden fallar. El patrón usado en el ejemplo de la función `contrived` es refutable. Existen otros tres tipos de patrones irrefutables, dos de los cuales introducimos a continuación (el otro lo postergamos hasta la Sección 4.4).

Patrones-nombrados.

Algunas veces es conveniente dar un nombre a un patrón para poder usarlo en la parte derecha de una ecuación. Por ejemplo la función que duplica el primer elemento de una lista podemos escribirla así:

```
f (x:xs)                = x:x:xs
```

(recordemos que "`:`" es asociativo a la izquierda.) Nótese que `x:xs` aparece como patrón en la parte izquierda y como expresión en la parte derecha. Para mejorar la lectura, desearíamos escribir `x:xs` una sola vez, lo cual se consigue utilizando un *patrón-nombrado* como sigue: (Otra ventaja es relativa a la implementación ya que en lugar de reconstruir completamente `x:xs` se usa el valor comparado.)

```
f s@(x:xs)              = x:s
```

Técnicamente hablando, un patrón-nombrado siempre tiene éxito, aunque el patrón que nombre pueda fallar.

Comodines (Wild-cards).

Otra situación corriente es comparar con un valor que realmente no necesitamos. Por ejemplo, las funciones `head` y `tail` definidas en la Seción 2.1 pueden reescribirse como:

```
head (x:_)              = x
tail (_:xs)             = xs
```

en las cuales hemos "avisado" del hecho de que no necesitamos cierta parte del argumento de entrada. Un comodín puede "emparejarse" con un valor arbitrario, pero en contraste con las variables, no se instancia a nada; por esta razón, se permiten varios comodines en la misma ecuación.

4.1 Semántica de la Comparación de Patrones

Vimos cómo son comparados los patrones individuales, cómo algunos son refutables y cómo otros son irrefutables, etc. Pero, ¿cómo se activa el proceso global? ¿en qué orden se intentan las comparaciones? ¿qué ocurre si ninguna tiene éxito? En esta sección discutimos estas cuestiones.

La comparación de patrones puede *fallar*, puede *tener éxito*, o puede *divergir*. Una comparación exitosa instancia (liga) los parámetros formales del patrón. La divergencia ocurre cuando un valor necesario es evaluado a error (_|_). El proceso de comparación se produce "hacia-abajo y de izquierda a derecha". El fallo de un patrón en algún lugar de una ecuación produce un fallo de toda la ecuación, y entonces se intenta con la siguiente ecuación. Si todas las ecuaciones fallan, el valor de la aplicación de la función es _|_, y entonces se provoca un error en tiempo de ejecución.

Por ejemplo, si comparamos `[1,2]` con `[0,bot]`, entonces `1` no encaja con `0`, y el resultado en un fallo. (Notemos que `bot`, definido anteriormente, es una variable con valor _|_.) Pero si comparamos `[1,2]` con `[bot,0]`, entonces la comparación de `1` con `bot` produce divergencia (i.e. _|_).

Una excepción a este conjunto de reglas es el caso en que los patrones (globales) puedan tener una *guarda* booleana, como en la siguiente definición de función que implementa una versión abstracta de la función signo:

```
sign x |  x >  0        =   1
       |  x == 0        =   0
       |  x <  0        =  -1
```

Nótese que podemos proporcionar una secuencia de guardas para el mismo patrón, y como cón éstos, las guardas son evaluadas hacia abajo, y la primera que se evalue a `True` produce un éxito en la comparación.

4.2 Un Ejemplo

Las reglas de la comparación de patrones pueden tener efectos delicados sobre el significado de las funciones. Por ejemplo, consideremos la siguiente definición de `take`:

```
take  0     _          =  []
take  _     []         =  []
take  n     (x:xs)     =  x : take (n-1) xs
```

y la siguiente versión ligeramente diferente (donde hemos invertido el orden en las dos primeras ecuaciones):

```
take1  _     []        =  []
take1  0     _         =  []
take1  n    (x:xs)     =  x : take1 (n-1) xs
```

Notemos entonces las siguientes reducciones:

```
take  0 bot  => []

take1 0 bot  => _|_

take  bot [] => _|_

take1 bot [] => []
```

Vemos que `take` está "más definida" con respecto a su segundo argumento, mientras `take1` está más definida con respecto a su primer argumento. Es difícil decir cual de las dos definiciones es mejor. Solo diremos que en ciertas aplicaciones podremos diferenciarlas. (El Standard Prelude incluye la definición correspondiente a `take`.)

4.3 Expresiones Case

La comparación de patrones proporciona una forma de "transferir el control" basada en las propiedades estructurales de un valor. Sin embargo, en varias situaciones no queremos definir una *función* para este fin, sino solamente especificar cómo realizar la comparación de patrones en la definición de funciones. Las *expresiones case* de Haskell proporcionan una forma para resolver este problema. En efecto; en el Informe se especifica el significado de la comparación de patrones a través de expresiones primitivas "case". En particular, una definición de función de la forma:

f p_{11} ... p_{1k} = e_1

...

f p_{n1} ... p_{nk} = e_n

donde cada p_{ij} es un patrón, es equivalente semánticamente a:

```
f x1 x2 … xk = case (x1, …, xk) of

(p11, …, p1k) -> e1

…

(pn1, …, pnk) -> en
```

donde los `xi` son identificadores nuevos. (Una traslación más general, que incluya construcciones con guardas, puede verse en §4.4.3.) Por ejemplo, la definición de `take` anterior es equivalente a:

```
take m ys               = case (m,ys) of
                            (0,_)       ->  []
                            (_,[])      ->  []
                            (n,x:xs)    ->  x : take (n-1) xs
```

Un aspecto no apuntado anteriormente es que por cuestiones de corrección de tipos, los tipos de los miembros derechos de una expresión "case" deben ser iguales; más exactamente, éstos deben tener el mismo tipo principal.

Las reglas de la comparación de patrones para las expresiones "case" son las mismas que para las definiciones de funciones, de forma que realmente no aportamos nada nuevo, salvo la conveniencia que ofrecen las expresiones "case".

Por el contrario, existe un uso de las expresiones case con una sintaxis especial: las *expresiones condicionales*. En Haskell, las expresiones condicionales tienen la forma familiar:

`if` e_1 `then` e_2 `else` e_3

que realmente es una forma abreviada de:

```
case e1 of True  -> e2

         False -> e3
```

A partir de este convenio, queda claro que e_1 debe ser de tipo `Bool`, mientras que e_2 y e_3 deben tener el mismo tipo principal (aunque arbitrario). En otras palabras, `if_then_else`_puede verse como una función de tipo `Bool->a->a->a`.

4.4 Patrones Perezosos

Existe otra clase de patrones permitidos en Haskell llamados *patrones perezosos*, y tienen la forma ~pat. Estos son *irrefutables*: la comparación de un valor v con ~pat siempre tiene éxito, independientemente de la expresión pat. Operacionalmente hablando, si, posteriormente, un identificador de pat es "utilizado" en la parte derecha de la ecuación,

éste será instanciado con la subexpresión del valor si la comparación de patrones tiene éxito; en otro caso, se instanciará con el valor _|_.

Los patrones perezosos son útiles en contextos donde aparecen estructuras de datos infinitas definidas recursivamente. Por ejemplo, las listas infinitas (usualmente llamadas *streams*) permiten describir en forma elegante programas de *simulación*. Como ejemplo elemental, consideremos la simulación de la interacción entre un proceso servidor `server` y un proceso cliente `client`, donde `client` envía una secuencia de *peticiones* a `server`, y `server` contesta a cada petición con alguna forma de *respuesta*. Esta situación se muestra en la Figura 2, al igual que para el ejemplo de la sucesión de Fibonacci. (Nótese que `client` también tiene un mensaje inicial como argumento.)

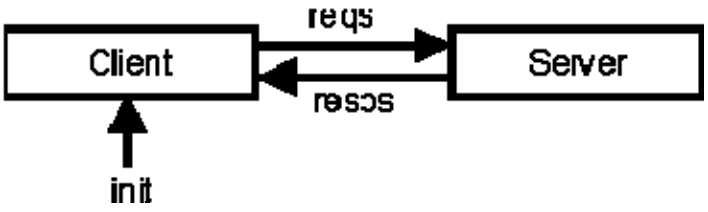

Figura 2

Utilizando streams para representar la secuencia de mensajes, el código Haskell correspondiente a éste diagrama es:

```
reqs                    = client init resps
resps                   = server reqs
```

Tales ecuaciones recursivas son una traducción directa del diagrama.

Además podemos suponer que la estructura del servidor y del cliente son de la forma siguiente:

```
client init (resp:resps) = init : client (next resp) resps
server      (req:reqs)   = process req : server reqs
```

donde suponemos que `next` es una función que, a partir de una respuesta desde el servidor, determina la siguiente petición, y `process` es una función que procesa una petición del cliente, devolviendo la respuesta apropiada.

Desafortunadamente, este programa tiene un problema serio: ¡no produce ninguna salida! El problema es que `client`, como es usada recursivamente en la inicialización de `reqs` y `resps`, espera una comparación con la lista de respuestas antes de enviar la primera petición. En otra palabras, la comparación de patrones se realiza "demasiado pronto." Una forma de resolver el problema es redefinir `client` como sigue:

```
client init resps       = init : client (next (head resps)) (tail resps)
```

A pesar de que funciona, esta solución no es tan expresiva como la anterior. Una forma mejor de resolver el problema es a través de un patrón perezoso:

```
client init ~(resp:resps) = init : client (next resp) resps
```

Puesto que los patrones perezosos son irrefutables, la comparación tiene éxito de

inmediato, permitiéndose que la petición inicial sea "enviada," y a su vez permitiendo que se genere la primera respuesta; de esta forma "arrancamos" el motor, y la recursión hace el resto.

Un ejemplo de cómo actua este programa es el siguiente:

```
init                    = 0
next resp               = resp
process req             = req+1
```

en el cual vemos que:

```
take 10 reqs => [0,1,2,3,4,5,6,7,8,9]
```

Como otro ejemplo de uso de patrones perezosos, consideremos la definición de Fibonacci dada anteriormente:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

Podemos intentar reescribirla usando un patrón nombrado:

```
fib@(1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

Esta versión de `fib` tiene la (pequeña) ventaja de que no usa `tail` a la derecha, puesto que aparece disponible en una forma no "destructiva" en el miembro de la izquierda a través del identificador `tfib`.

[Este tipo de ecuaciones se llama un *enlace a través de patrones* (*pattern binding*); éstas son aquellas ecuaciones donde la parte izquierda completa es un patrón; obsérvese que tanto `fib` como `tfib` aparecen ligados en el ámbito de la declaración.]

Ahora, utilizando el mismo razonamiento que antes, deberíamos concluir que este programa no genera ninguna salida. Sin embargo, cursiosamente *funciona*, y la razón es simple: en Haskell se supone que el enlace a través de patrones tiene un carácter ~ implícito delante, reflejando el comportamiento esperado, y evitando algunas situaciones anómalas que están fuera del alcance de este tutorial. De esta forma, vemos que los patrones perezosos juegan un papel importante en Haskell, y de forma implícita.

4.5 Ambito léxico y declaraciones locales

A menudo es deseable generar un ámbito dentro de una expresión, con el objetivo de crear ligaduras o definiciones locales no "vistas fuera"---es decir, alguna forma de "estructuración por bloques". Para ello tenemos dos formas en Haskell:

Expresiones Let.

Las *expresiones let* de Haskell son útiles cuando se requiere un conjunto de declaraciones locales. Un ejemplo simple es el siguiente:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

El conjunto de definiciones creada por una expresión `let` es *mutuamente recursiva*, y la comparación de patrones es tratada en forma perezosa (es decir, tienen un ~ implícito). La únicas declaraciones permitidas son *declaraciones de tipos de funciones*, *enlace de funciones*, y *enlace de patrones*.

Cláusulas Where.

A veces es conveniente establecer declaraciones locales en una ecuación con varias con guardas, lo que podemos obtener a través de una *cláusula where*:

```
f x y  |  y>z           =  ...
       |  y==z          =  ...
       |  y<z           =  ...
     where z = x*x
```

Nótese que ésto no puede obtenerse con una expresión `let`, la cual únicamente abarca a la expresión que contiene. Solamente se permite una cláusula `where` en el nivel externo de un grupo de ecuaciones o expresión case. Por otro lado, las cláusulas `where` tienen las mismas propiedades y restricciones sobre las ligaduras que las expresiones `let`.

Estas dos formas de declaraciones locales son muy similares, pero recordemos que una expresión `let` es una *expresión*, mientras que una cláusula `where` no ---forma parte de la sintaxis de la declaración de funciones y de expresiones case.

4.6 Sangrado (layout)

El lector puede preguntarse cómo es que los programas Haskell evitan el uso de punto-coma o algún otro tipo de separador con objeto de determinar el final de las ecuaciones, declaraciones, etc. Por ejemplo, consideremos la expresión `let` de la última sección:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

¿Cómo distingue el analizador entre el código anterior y el siguiente:?

```
let y   = a*b f
    x   = (x+y)/y
in f c + f d
```

La respuesta es que Haskell usa una sintaxis bi-dimensional llamada *layout* que esencialmente presupone que las declaraciones están "alineadas por columnas." En el ejemplo primero, nótese que tanto `y` como `f` comienzan en la misma columna. Las reglas aparecen con detalle en el Informe (§2.5,§B.3), pero en la práctica, el uso es intuitivo. Únicamente hay que recordar dos detalles:

En primer lugar, el siguiente carácter que siga a una de las palabras reservadas `where`, `let`, o `of`, determina la columna de comienzo donde deben escribirse las declaraciones locales correspondientes a where, let o case (esta regla también es aplicable a la cláusula `where` cuando se use en declaraciones de clases e instancias, que veremos en la Sección 5). Por tanto, podemos comenzar las declaraciones en la misma línea, o en la línea siguiente a la

palabra reservada, etc. (La palabra reservada `do`, que discutiremos más tarde, también usa esta regla).

En segundo lugar, debemos asegurarnos que la columna de comienzo aparece más a la derecha que la correspondiente a la declaración actual (en otro caso tendríamos ambigüedad). El "final" de una declaración se detecta cuando aparece algo a la izquierda de la columna de comienzo asociada. (Haskell presupone que los tabuladores cuentan como 8 espacios; este detalle debe tenerse en cuenta cuando usemos un editor con otro convenio.)

El "Layout" es realmente una forma abreviada de un mecanismo de agrupamiento *explícito* que debemos mencionar ya que puede ser útil en ciertas circunstancias. El ejemplo de expresión `let` anterior es equivalente a:

```
let { y   = a*b
    ; f x = (x+y)/y
    }
in f c + f d
```

Nótese las llaves y punto-coma. Tal notación explícita es útil si queremos colocar varias declaraciones en una sola línea; por ejemplo, la siguiente expresión es válida:

```
let y   = a*b;  z = a/b
    f x = (x+y)/z
in f c + f d
```

Otros ejemplos de uso del layout con delimitadores pueden verse en §2.7.

El uso del layout reduce en gran medida el desorden sintáctico asociado con los grupos de declaraciones, mejorando la legibilidad. Es fácil de aprender y recomendamos su uso.

5 Clases de tipos y Sobrecarga

El sistema de tipos de Haskell posee una característica que lo distingue de otros lenguajes de programación. El tipo de polimorfismo del que hemos tratado hasta ahora es denominado polimorfismo *paramétrico*. Existe otro tipo de polimorfismo llamado *ad hoc* o *sobrecarga*. Estos son algunos ejemplos de polimorfismo ad hoc:

- Los literales `1`, `2`, etc. son usados para representar tanto valores enteros de precisión arbitraria, como enteros de precisión limitada.
- Los operadores numéricos, como +, suelen estar definidos para distintos tipos numéricos.
- El operador de igualdad (== en Haskell) suele estar definido para bastantes tipos de datos (como los numéricos y caracteres) aunque no para todos.

Obsérvese que el comportamiento de estas funciones u operadores sobrecargados suele ser distinto para cada tipo de dato (de hecho, el comportamiento está indefinido o es erróneo en ciertos casos), mientras que en el caso del polimorfismo paramétrico, el comportamiento de la función no depende del tipo (el comportamiento de `fringe`, por ejemplo, no depende del tipo de los elementos almacenados en las hojas del árbol). En Haskell, las *clases de tipos* proporcionan un modo estructurado de controlar el polimorfismo ad hoc o sobrecarga.

Empecemos considerando un ejemplo simple pero a la vez importante: la igualdad. Hay muchos tipos para los cuales nos gustaría que la igualdad estuviese definida, y algunos para los que no. Por ejemplo, la determinación de la igualdad de funciones es, en general, considerada no computable, mientras que la comparación de listas es algo habitual. (El tipo de igualdad que estamos considerando es "comparación de valores", y es distinto a la igualdad de punteros que aparece en otros lenguajes, como el operador == de Java. La igualdad de punteros rompe la transparencia referencial y, por tanto, no es apropiada para un lenguaje funcional puro). Como ejemplo práctico, considérese esta definición de la función `elem` que comprueba la pertenencia a una lista:

```
x `elem`  []            = False
x `elem` (y:ys)         = x==y || (x `elem` ys)
```

[Debido a la norma de estilo discutida en la sección 3.1, hemos definido `elem` de forma infija. `==` y `||` son operadores infijos para la igualdad y la disyunción lógica, respectivamente.]

De un modo intuitivo, el tipo de `elem` "debería" ser: `a->[a]->Bool`. Pero esto implicaría que == tuviese como tipo `a->a->Bool`, aunque acabamos de decir que no esperamos que == esté definido para todos los tipos.

Además, como hemos destacado anteriormente, incluso si == estuviese definido para cualquier tipo, el cómputo realizado para comparar dos listas es distinto al realizado al comparar dos enteros. Por todo ello, es de esperar que == esté *sobrecargado* para realizar estas tareas distintas.

Las *clases de tipos* resuelven todos estos problemas de un modo adecuado. Permiten declarar qué tipos son *instancias* de qué clases, y dar definiciones para las *operaciones*

sobrecargadas asociadas con cada clase. Por ejemplo, definamos una clase de tipos conteniendo el operador de igualdad:

```
class Eq a where
  (==)                  :: a -> a -> Bool
```

Donde `Eq` es el nombre de la clase definida, y == es la única operación en la clase. Esta declaración puede ser leída como "un tipo arbitrario `a` es una instancia de la clase `Eq` si existe una operación sobrecargada ==, del tipo adecuado, definida para él." (Obsérvese que == solo está definido sobre pares de objetos del mismo tipo.)

La restricción de que un tipo `a` debe ser una instancia de la clase `Eq` se escribe `Eq a`. Así, `Eq a` no es una expresion de tipo, sino que expresa una restricción sobre un tipo, y se denomina un *contexto*. Los contextos aparecen al inicio de las expresiones de tipo. Por ejemplo, el resultado de la declaración de clase anterior es asignar el siguiente tipo a ==:

```
(==)                    :: (Eq a) => a -> a -> Bool
```

Este tipo debe ser leído del siguiente modo: "Para cada tipo `a` que es una instancia de la clase `Eq`, == tiene como tipo `a->a->Bool`." Este es el tipo que es usado para == en la definición de `elem` del ejemplo, y de hecho, la restricción impuesta por el contexto es propagada al tipo principal de `elem`:

```
elem                    :: (Eq a) => a -> [a] -> Bool
```

El tipo anterior debe ser leído como, "Para cada tipo `a` que sea una instancia de la clase `Eq`, `elem` tiene por tipo `a->[a]->Bool`." Esto es exactamente los que queríamos---expresar que `elem` no está definido para todos los tipos, sino solo para aquellos que pueden ser comparados con el operador de igualdad.

Todo esto está muy bien, pero ¿cómo especificamos qué tipos son instancias de la clase `Eq`, y el comportamiento de == para cada uno de ellos?. Esto se consigue con una *declaración de instancia*. Por ejemplo:

```
instance Eq Integer where
  x == y                =  integerEq x y
```

== es denominado un *método* de la clase. La función `integerEq` es una primitiva que compara dos enteros, aunque, en general, cualquier expresión válida puede aparecer en la parte derecha de la declaración, como en cualquier otra definición de función. Toda la declaración viene a decir: "El tipo `Integer` es una instancia de la clase `Eq`, y esta es la definición del método correspondiente a la operación == para los enteros." Dada esta declaración, podemos comparar enteros de precisión limitada usando ==. De un modo similar:

```
instance Eq Float where
  x == y                =  floatEq x y
```

permite comparar números reales usando ==.

Los tipos recursivos definidos previamente, como `Tree`, también pueden ser tratados:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a         == Leaf b         =  a == b
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1==r2)
  _              == _              =  False
```

Obsérvese el contexto `Eq a` en la primera línea---esto es necesario porque los elementos en las hojas (que tienen tipo `a`) son comparados en la parte derecha de la segunda línea. La restricción adicional viene a decir esencialmente que podemos determinar la igualdad de árboles cuyos elementos tienen tipo `a` siempre que podamos determinar la igualdad de datos de tipo `a`. Si los contextos hubiesen sido omitidos en la declaración de instancia, se habría producido un error de tipo estático (en tiempo de compilación).

El Informe de Haskell, y en especial el Prelude, contiene bastantes ejemplos útiles de clases de tipo. La definición completa de la clase `Eq` es un poco más extensa:

```
class  Eq a  where
  (==), (/=)            :: a -> a -> Bool
  x == y                =  not (x /= y)
  x /= y                =  not (x == y)
```

Éste es un ejemplo de una clase con dos operaciones, una para determinar la igualdad y otra para la desigualdad. También demuestra el uso de los *métodos por defecto*, en este caso para los operadores `/=` y `==`. Si un método para una operación concreta es omitido en una declaración de instancia, entonces el definido por defecto en la declaración de clase, si existe, es usado. Por ejemplo, las tres instancias de `Eq` definidas previamente son correctas dada la nueva declaración de la clase `Eq`. Además, la definición de desigualdad es adecuada para los tipos considerados: la negación lógica de la igualdad.

Haskell también posee una noción de *extensión de clases*. Por ejemplo, podemos definir una clase `Ord` que *hereda* todas las operaciones de `Eq`, y que además posee operadores de orden y funciones para calcular el máximo y mínimo de dos valores:

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

Obsérvese el contexto en la declaración de `clase`. Decimos que `Eq` es una *superclase* de `Ord` (recíprocamente, `Ord` es una *subclase* de `Eq`). El significado de la extensión de clases es que cualquier tipo que sea una instancia de `Ord` debe ser también una instancia de `Eq`. (En la próxima sección daremos las definición completa de la clase `Ord` del Prelude.) El compilador producirá un error en caso contrario.

Uno de los beneficios de esta jerarquía de clases es que los contextos de la funciones pueden ser abreviados: una expresión de tipo para una función que usa operaciones tanto de la clase `Eq` como de `Ord` puede usar simplemente el contexto `(Ord a)`, en vez de `(Eq a, Ord a)`, ya que `Ord` "implica" `Eq`. Más importante es el hecho de que los métodos para operaciones de subclases pueden asumir la existencia de métodos para las operaciones de la superclase. Por ejemplo, la declaración de la clase `Ord` del Prelude contiene esta definición por defecto para `(<)`:

```
    x < y               =  x <= y && x /= y
```

Como ejemplo del uso de la clase `Ord`, el tipo principal de la función `quicksort` definida en la sección 2.4.1 es:

```
quicksort               ::  (Ord a) => [a] -> [a]
```

De otro modo: `quicksort` sólo opera con listas de valores ordenables. El tipo de `quicksort` es debido al uso de los operadores de comparación < y >= en su definición.

Haskell también permite *herencia múltiple*, ya que una clase puede tener más de una superclase. Los conflictos de nombres son evitados con la restricción de que una operación solo puede ser miembro de una única clase en un mismo ámbito. Por ejemplo, la declaración

```
class (Eq a, Show a) => C a where ...
```

crea una clase `C` que hereda las operaciones de `Eq` y `Show`.

Los métodos de clase son tratados como declaraciones globales en Haskell. Comparten el mismo espacio de nombres que las variables normales; así, un mismo nombre no puede ser usado para denotar a la vez un miembro de clase y una variable o métodos en distintas clases.

Los contextos también pueden ser usados en las declaraciones de tipos (`data`); véase §4.2.1.

Los métodos de clases pueden tener restricciones adicionales sobre cualquier variable de tipo excepto aquella que aparece en la declaración de clase tras el nombre de la clase definida. Por ejemplo, en esta clase:

```
class C a where
  m :: Show b => a -> b
```

el método `m` requiere que el tipo `b` esté en la clase `Show`. Sin embargo, no es posible establecer un contexto adicional sobre el tipo `a` en el método `m`. Para ello, la restrición debería ser parte del contexto en la declaración de la clase.

Hasta ahora, hemos usado tipos de "primer orden". Por ejemplo, el constructor de tipos `Tree` ha aparecido siempre seguido de un argumento, como en `Tree Integer` (un árbol que almacena valores de tipo `Integer`) or `Tree a` (representando la familia de tipos de árboles conteniendo valores de tipo `a`). Pero `Tree` por sí mismo es un constructor de tipos, y como tal toma un tipo como argumento y devuelve un tipo como resultado. No hay ningún valor en Haskell que tenga como tipo simplemente `Tree`, pero los "tipos de orden superior" pueden ser usados en las declaraciones de clase.

Para empezar, consideremos la siguiente clase `Functor` (tomada del Prelude):

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

La fución `fmap` generaliza a la función `map` usada previamente. Obsérvese que la variable de tipo `f` está siendo aplicada a otro tipo en `f a` y `f b`. Así que podemos esperar que

aparezca ligada a un tipo como `Tree` que puede ser aplicado a un argumento. Una instancia de la clase `Functor` para el tipo `Tree` puede ser:

```
instance Functor Tree where
  fmap f (Leaf x)       = Leaf   (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Esta declaración de instacia indica que `Tree`, en vez de `Tree a`, es una instancia de la clase `Functor`. Esta característica del lenguaje es muy útil, y demuestra la posibilidad de definir tipos contenedores (*containers*), que implementen funciones como `fmap`, de modo que ésta funcione de un modo uniforme sobre árboles, listas y otros tipos de datos.

[La aplicación de tipos se escribe del mismo modo que la aplicación de funciones. El tipo `T a b` es analizado sintácticamente como `(T a) b`. Algunos tipos como las tuplas, que usan una sintaxis especial, pueden ser escritos de un modo alternativo que permite la currificación. El constructor para los tipos función es `(->)`; Los tipos `f -> g` y `(->) f g` son idénticos. De un modo similar, los tipos `[a]` y `[] a` son idénticos también. Para las tuplas, los constructores de tipo (del mismo modo que los constructores de datos) son `(,)`, `(,,)`, y así sucesivamente.]

Como ya sabemos, el sistema de tipos detecta los errores de tipo en las expresiones. Pero, ¿qué es lo que ocurre con las expresiones de tipo malformadas ? La expresión `(+) 1 2 3` da lugar a un error de tipos ya que `(+)` toma tan solo dos argumentos. De un modo similar, el tipo `Tree Integer Integer` debería provocar alguna clase de error ya que el tipo `Tree` toma tan solo un único argumento. Entonces, ¿Cómo detecta Haskell expresiones de tipo erróneas ? La respuesta es que utiliza un segundo sistema de tipos que asegura la correción de los tipos. Cada tipo tiene un *género* (*kind*) asociado que puede ser usado para asegurar que el tipo es usado de modo correcto.

Las expresiones de tipo son clasificadas según sus *géneros* los cuales pueden tomar una de las siguientes formas:

- El símbolo * representa el género de tipos asociados con datos concretos. Es decir, si el valor *v* tiene tipo *t*, el género de *v* debe ser *.
- Si κ_1 y κ_2 son géneros, entonces κ_1->κ_2 es el género de los tipos que toman un tipo con género κ_1 y devuelven un tipo con género κ_2.

El constructor de tipos `Tree` tiene como género *->*; el tipo `Tree Integer` tiene género *. Los miembros de la clase `Functor` han de tener todos el género *->*; a partir de la siguiente declaración, el compilador generaría un error de géneros

```
instance Functor Integer where ...
```

ya que `Integer` tiene como género *.

Los géneros no aparecen explícitamente en los programas escritos en Haskell. El compilador infiere los géneros antes de realizar el chequeo de tipos, sin que sea necesario (ni posible) introducir en el programa "declaraciones de género". Los géneros permanecen en un segundo plano excepto cuando se produce un error de género. Los géneros son lo suficientemente simples como para que un compilador pueda producir un mensaje de error

descriptivo en caso de que se produzca. Véase §4.1.2 y §4.6 para obtener más información sobre los géneros.

Una perspectiva diferente.

Antes de mostrar más ejemplos de uso de clases, merece la pena resaltar otros dos modos de ver las clases de Haskell. El primero es la analogía con la programación orientada a objetos (POO). Si sustituimos clase por *clase de tipos* y objeto por *tipo* en las siguientes afirmaciones sobre POO, obtendremos un resumen válido de qué representan las clases de tipos de Haskell:

"Las *clases* capturan conjuntos comunes de *operaciones*. Un *objeto* concreto puede ser una *instancia* de una clase, y habrá un *método* para cada operación. Las clases pueden aparecer jerarquizadas, dando lugar a las nociones de *superclase* y *subclase*, y permitiendo la *herencia* de operaciones/métodos. Un *método por defecto* puede estar asociado con una operación."

Al *contrario* que en POO, los tipos no son objetos, y en concreto, no tiene sentido la noción de estado modificable interno de un objeto o tipo. Una ventaja en relación con algunos lenguajes orientados a objetos es que los métodos de Haskell son completamente seguros con respecto a los tipos (*type-safe*): cualquier intento de aplicar un método a un valor cuyo tipo no esté en la clase requerida será detectado en tiempo de compilación en vez de en tiempo de ejecución. Es decir, los métodos no son buscados en tiempo de ejecución, sino que son simplemente pasados como argumentos a funciones de orden superior.

Una perspectiva diferente es considerar la relación entre el polimorfismo paramétrico y la sobrecarga. Hemos visto que el polimorfismo paramétrico es útil a la hora de definir familias de tipos al cuantificar universalmente sobre todos los tipos. A veces, sin embargo, esa cuantificación universal es demasiado amplia ---nos gustaría cuantificar sobre un conjunto de tipos menor, como, por ejemplo, aquellos tipos para cuyos elementos la igualdad puede ser determinada. Las clases de tipos proporcionan un modo estructurado de hacer esto. De hecho, podemos ver también el polimorfismo paramétrico como una forma de sobrecarga. La sobrecarga se realiza sobre todos los tipos en vez de sobre un conjunto restringido de éstos (sobre una clase de tipos).

Comparación con otros lenguajes.

Las clases de Haskell son similares a las usadas en algunos lenguajes orientados a objetos como C++ y Java. Sin embargo, hay diferencias significativas:

- Haskell separa la definición de un tipo de la definición de los métodos asociados con dicho tipo. Una clase en C++ o Java suele definir tanto una estructura de datos (las variables miembros) como las operaciones asociadas con la estructura (los métodos). En Haskell, estas definiciones aparecen separadas.
- Los métodos de clase definidos en una clase Haskell se corresponden con las funciones virtuales de C++. Cada instancia de una clase proporciona su propia definición para cada método; las métodos por defecto se corresponden con las definiciones por defecto de una función virtual en una clase base.

- Las clases de Haskell son similares a los interfaces de Java a grandes rasgos. Al igual que una declaración de interfaz, una clase de Haskell define un protocolo para usar un objeto en vez de un objeto en sí.
- Haskell no proporciona el estilo de sobrecarga de C++, en el cual, funciones con tipos distintos pueden compartir un mismo nombre.
- El tipo de un objeto Haskell no puede ser promocionado (*coerced*) implícitamente; no hay una clase base universal tal como `Object` cuyos valores pueden ser projectados a otros objetos.
- C++ y Java incluyen información identificativa (como la *VTable*) en la representación en tiempo de ejecución de un objeto. En Haskell, esa información es adjuntada de un modo lógico en vez de físico gracias al sistema de tipos.
- El sistema de clases de Haskell no contempla el control de acceso a los métodos (tales como accesos privados o públicos). En Haskell, el sistema de módulos puede ser usado para ocultar o revelar los componentes de una clase.

6 Más sobre Tipos

Pasamos a examinar algunos de los aspectos más avanzados de las declaraciones de tipo.

6.1 La declaración newtype

Una práctica común en programación es definir un tipo cuya representación es idéntica a otro tipo existente pero que tenga una identidad propia para el sistema de tipos. En Haskell, la declaración `newtype` crea un nuevo tipo a partir de uno existente. Por ejemplo, los números naturales pueden ser representados usando el tipo `Integer` mediante la siguiente declaración:

```
newtype Natural = MakeNatural Integer
```

Esta declaración crea un tipo completamente nuevo, `Natural`, cuyo único constructor permite almacenar un `Integer`. El constructor `MakeNatural` permite convertir un dato de tipo `Integer` en uno de tipo `Natural`:

```
toNatural :: Integer -> Natural
toNatural x | x < 0     = error "No es posible crear naturales
negativos!"
            | otherwise = MakeNatural x

fromNatural :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

La siguiente declaración de instancia convierte al tipo `Natural` en instancia de la clase `Num`:

```
instance Num Natural where
    fromInteger = toNatural
    x + y = toNatural (fromNatural x + fromNatural y)
    x - y = let r = fromNatural x - fromNatural y in
              if r < 0 then error "Sustracción no natural"
                       else toNatural r
    x * y = toNatural (fromNatural x * fromNatural y)
```

Sin esta declaración, el tipo `Natural` no sería de la clase `Num`. Las instancias declaradas para el tipo `Integer` no dan lugar automáticamente a las correspondientes instancias para el tipo `Natural`. De hecho, el propósito principal de este tipo es introducir una instancia de la clase `Num` diferente. Esto no habría sido posible si el tipo `Natural` hubiese sido definido como un sinónimo del tipo `Integer`.

Todo lo comentado funciona si usamos una declaración `data` en vez de una declaración `newtype`. Sin embargo, la declaración `data` da lugar a un coste extra en la representación de valores del tipo `Natural`. El uso de `newtype` evita el nivel extra de indirección (causado por la evaluación perezosa) que la declaración `data` introduciría. Véase la sección 4.2.3 del informe del lenguaje para una discusión en profundidad de las declaraciones `newtype`, `data`, y `type`. [Excepto por la palabra clave inicial, las declaraciones `newtype` usan la misma sintaxis que las declaraciones `data` que poseen un único constructor con un único

campo. Esto es consistente, ya que los tipos definidos usando `newtype` son casi idénticos a los declarados usando una declaración `data` ordinaria.]

6.2 Identificadores de componentes

Los campos de un dato pueden ser accedidos tanto por su posición como por su nombre, usando en este último caso, las *etiquetas de campo*. Considérese el siguiente tipo para definir un punto bidimiensional:

```
data Point = Pt Float Float
```

Las dos componentes del punto son el primer y segundo argumento del constructor `Pt` respectivamente. Una función como

```
pointx          :: Point -> Float
pointx (Pt x _) =  x
```

puede ser usada para acceder a la primera componente del punto de un modo más descriptivo, pero para estructuras de datos más extensas, resulta pesado definir dichas funciones manualmente.

Los constructores en una declaración `data` pueden contener *nombres de campo* colocándolos entre llaves. Estos nombres de campo permiten identificar los componentes de un constructor por su nombre en vez de por su posición. Una definición alternativa del tipo `Point` es:

```
data Point = Pt {pointx, pointy :: Float}
```

Esta declaración de tipo es idéntica a la definición previa del tipo `Point`. El constructor `Pt` es el mismo en ambos casos. Sin embargo, esta declaración también introduce dos nombres de campo, `pointx` y `pointy`. Estos nombres de campo pueden ser usados como *funciones selectoras* para extraer un componente de la estructura. Para el ejemplo considerado, los selectores son:

```
pointx   ::   Point -> Float
pointy   ::   Point -> Float
```

La siguiente función usa estos selectores:

```
absPoint  :: Point -> Float
absPoint p = sqrt (pointx p * pointx p + pointy p * pointy p)
```

Las etiquetas de campo pueden ser usadas también para construir nuevos valores. La expresión `Pt {pointx=1, pointy=2}` es idéntica a `Pt 1 2`. El uso de nombres de campo en la declaración de un constructor de datos no prohibe el modo posicional de acceso a los campos; tanto `Pt {pointx=1, pointy=2}` como `Pt 1 2` son válidos. Es posible omitir algún campo al definir valores usando nombres de campo; los campos ausentes estarán indefinidos.

Los patrones para las etiquetas de campo tienen una sintaxis parecida para el constructor `Pt`:

```
absPoint (Pt {pointx = x, pointy = y}) = sqrt (x*x + y+y)
```

Una función de actualización usa los nombres de campo existentes en una estructura para rellenar los componentes de otra. Si `p` es un valor de tipo `Point`, entonces `p {pointx=2}` es un punto con la misma componente `pointy` que `p` pero con el valor 2 en la componente `pointx`. Esta operación no es una asignación destructiva: la función de actualización simplemente crea una nueva copia del objeto, rellenando los campos especificados con nuevos valores.

[Las llaves usadas junto con las etiquetas de campo son especiales en cierto modo: la sintaxis de Haskell suele permitir que las llaves sean omitidas usando la regla del sangrado (*layout rule*, descrita en la sección 4.6). Por el contrario, las llaves asociadas con los nombres de campo deben aparecer explícitamente.]

Los nombres de campos no están restringidos a tipos con un constructor único (habitualmente llamados tipos "registro"). En un tipo con constructores múltiples, la selección o la actualización mediante nombres de campo puede dar lugar a un error en tiempo de ejecución. Esto es parecido al comportamiento de la función `head` al actuar sobre listas vacías.

Las nombres de campo comparten el mismo espacio de nombres que las variables normales y los métodos de clases. Un nombre de campo no puede ser usado en más de un tipo de datos en un mismo ámbito. Sin embargo, dentro de un mismo tipo de datos, el mismo nombre de campo puede ser usado en más de un constructor siempre que el tipo del campo sea el mismo en todos los casos. Por ejemplo, en la siguiente declaración de tipo

```
data T = C1 {f :: Int, g :: Float}
       | C2 {f :: Int, h :: Bool}
```

el nombre de campo `f` aparece en los dos contructores de `T`. Así, si `x` tiene tipo `T`, entonces `x {f=5}` es válido para valores creados a partir de cualquiera de los constructores de `T`.

Los nombres de campo no modifican la naturaleza básica de una estrucutura de datos algebraica; son simplemente una sintaxis apropiada que permite acceder a los componentes de una estructura de datos utilizando nombres en vez de posiciones. Hacen más manejable el uso de constructores con varias componentes, ya que los campos pueden ser añadidos o eliminados sin cambiar cada aparición del constructor. Para otros detalles de los nombres de campo y su semántica, véase la sección §4.2.1.

6.3 Constructores de datos estrictos

Las estructuras de datos de Haskell son normalmente perezosas: las componentes de éstas no son evaluadas hasta que sea necesario. Esto permite definir estructuras que contienen elementos cuya evaluación, si se produce, puede provocar un error o no terminar. Las estructuras de datos perezosas mejoran la expresividad de Haskell y son una parte fundamental de su estilo de programación.

Internamente, cada componente de una estructura de datos perezosa es envuelta en una estructura usualmente denominada cierre (*thunk*), que encapsula la computación que producirá el valor. Este cierre no es evaluado hasta que el valor es necesitado; los cierres que contienen errores (_|_) no afectan a otros componentes de la estructura. Por ejemplo, la tupla `('a',_|_)` es un valor totalmente legal en Haskell. El caracter `'a'` puede ser usado sin ningún problema. La mayoría de los lenguajes de programación son estrictos (*strict*) en vez de perezosos (*lazy*); en los lenguajes estrictos, todos los componentes de una estructura son reducidos a sus respectivos valores antes de construir la estructura.

Hay una serie de costos asociados a los cierres: es necesario tiempo para construirlos y para evaluarlos, ocupan espacio en el montículo (*heap*), y hacen que el recolector de basura (*garbage collector*) mantenga en memoria otras estructuras que pueden ser necesarias para una posible evaluación posterior del cierre. Para evitar estos costos, los indicadores de estrictez (*strictness flags*) en las declaraciones `data` permiten que campos específicos de un constructor sean evaluados inmediatamente, lo cual ofrece la posibilidad al programador de suprimir de un modo selectivo la perezosidad del lenguaje. Un campo marcado con `!` en una declaración `data` es evaluado cuando la estructura es creada en vez de producir la creación de un cierre. Hay varias situaciones en las que el uso de los indicadores de estrictez puede ser adecuado:

- En componentes de estructuras para los que se puede asegurar que serán evaluados en algún momento de la ejecución del programa.
- En componentes de estructuras cuya evaluación no es costosa y que no causen errores.
- En tipos para los que valores parcialmente definidos no tienen sentido.

Por ejemplo, la biblioteca de números complejos define el tipo `Complex` como:

```
data RealFloat a => Complex a = !a :+ !a
```

[obsérvese el caracter infijo del constructor `:+`.] Esta definición marca los dos componentes (la parte real e imaginaria) del número complejo como estrictas. Esta representación es más compacta pero hace que los números complejos con una única componente indefinida, `1 :+ _|_` por ejemplo, estén totalmente indefinidos (_|_). Dado que no hay necesidad real de hacer uso de números complejos parcialmente definidos, tiene sentido usar indicadores de estrictez para obtener una representación más eficiente.

Los indicadores de estrictez suelen ser usados para solucionar escapes de memoria (*memory leaks*): estructuras retenidas en memoria por el recolector de basura que ya no son necesarias para el cómputo restante.

El indicador de estrictez, `!`, sólo puede aparecer en declaraciones `data`. No pueden ser usados en otras declaraciones de tipo o en otras definiciones de tipo. No hay un modo de marcar argumentos de funciones como estrictos, aunque este efecto puede ser conseguido usando la función `seq` y el operador `$!`. Véase §4.2.1 para más detalles.

Es difícil presentar una guía general para el uso de los indicadores de estrictez. Deben ser usados con precaución: la perezosidad es una de las propiedades fundamentales de Haskell y las anotaciones pueden dar lugar a bucles infinitos difíciles de detectar o tener consecuencias inesperadas.

7 Entrada/Salida

El sistema de entrada/salida (E/S) de Haskell es funcional puro, y además, proporciona toda la expresividad que presentan los lenguajes de programación imperativos convencionales. En los lenguajes imperativos, los programas están formados por *acciones* que examinan y modifican el estado actual del *sistema*. Algunas acciones típicas son la lectura y modificación de variables globales, la escritura en ficheros, la lectura de datos y el manejo de ventanas en entornos gráficos. Haskell permite utilizar acciones de este tipo, aunque están claramente separadas del núcleo puramente funcional del lenguaje.

El sistema de E/S de Haskell está basado en un fundamento matemático que puede asustar a primera vista: las *mónadas*. Sin embargo, no es necesario conocer la teoría de mónadas subyacente para programar usando el sistema de E/S. Más bien, las mónadas son simplemente una estructura conceptual en las que las E/S encaja. No es necesario conocer la teoría de mónadas para trabajar con operaciones de E/S en Haskell del mismo modo que no es necesario conocer la teoría de grupos para trabajar con operaciones aritméticas simples. Una explicación en profundidad de las mónadas puede encontrarse en la sección 9.

Los operadores monádicos en los que el sistema de E/S está basado son usados también para otros propósitos; profundizaremos en las mónadas posteriormente. Por ahora, evitaremos el término mónada y nos concentraremos en el uso del sistema de E/S. Es mejor pensar en la mónada de E/S como un tipo abstracto.

Las acciones son definidas, pero no invocadas, al nivel de la máquina de evaluación que dirige la ejecución de un programa Haskell. La evaluación de la definición de una acción no hace que la acción sea realizada. Más bien, la ejecución de acciones es efectuada en un nivel distinto a la evaluación de expresiones que hemos considerado hasta este momento.

Las acciones son atómicas, como el caso de las definidas como primitivas del sistema, o se obtienen de la composición secuencial de otras acciones. La mónada de E/S contiene primitivas que permiten construir acciones compuestas, un proceso similar al que realizan algunos lenguajes imperativos al unir sentencias de un modo secuencial usando ";". Esta mónada actúa como un pegamento que une las acciones que forman parte de un programa.

7.1 Operaciones de E/S básicas

Cada acción de E/S devuelve un valor. A nivel del sistema de tipos, el valor devuelto es anotado con un tipo `IO`. Esto distingue las acciones de otros valores. Por ejemplo, el tipo de la función `getChar` es:

```
getChar    ::   IO Char
```

El tipo `IO Char` indica que `getChar`, cuando sea ejecutado, realizará una acción que devolverá un carácter. Las acciones cuyo valor devuelto no es interesante usan el tipo unitario `()`. Por ejemplo, la función `putChar`:

```
putChar    ::    Char -> IO ()
```

toma un carácter como argumento pero no devuelve nada útil. El tipo unitario es similar al tipo `void` en otros lenguajes.

Las acciones son combinadas secuencialmente usando un operador cuyo nombre es un tanto críptico: `>>=` (o "*bind*"). En vez de usar este operador directamente, usaremos una sintáxis más clara, la notación `do`, para ocultar el uso de los operadores de secuenciación gracias al uso de una sintaxis que recuerda a la de los lenguajes imperativos convencionales. La notación `do` puede ser traducida, de un modo trivial, a funciones Haskell normales, como se describe en §3.14.

La palabra clave `do` inicia una secuencia de sentencias que serán ejecutadas en orden. Una sentencia es o bien una acción, o un patrón asociado al resultado de una acción usando `<-`, o una definición local definida mediante `let`. La notación `do` usa la indentación del programa del mismo modo que las definiciones locales introducidas con `let` o `where`, de modo que podemos omitir las llaves y los puntos y coma usando un sangrado adecuado. El siguiente ejemplo es un programa que lee y escribe un carácter:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

El uso del nombre `main` es importante: `main` es la expresión principal de un programa Haskell (de un modo similar a la función `main` de un programa en C), y debe tener un tipo `IO`m, usualmente `IO ()`. (El nombre `main` es especial tan solo en el módulo `Main`; hablaremos de los módulos posteriormente). Este programa ejecuta dos acciones en secuencia: primero lee un carácter del teclado, ligando el resultado con la variable `c`, y a continuación imprime el carácter. A diferencia de las expresiones `let` para las que las variables introducidas están en el ámbito de todas las definiciones, las variables definidas con `<-` solo están en el ámbito de las sentencias posteriores.

Todavía falta algo. Podemos ejecutar acciones y examinar sus resultados usando la notación `do`, pero ¿cómo devolvemos un valor desde una secuencia de acciones? Por ejemplo, considérese la función `ready` que lee un carácter y devuelve `True` si dicho carácter es `y':

```
ready   :: IO Bool
ready   =  do c <- getChar
              c == 'y'  -- Mal !!!
```

La definición anterior no es correcta porque la segunda sentencia es un valor booleano, y no una acción. Es necesario crear una acción que devuelva un booleano como resultado. La función `return` hace precisamente esto:

```
return ::   a -> IO a
```

La función `return` completa el conjunto de primitivas. La última línea de la definición de `ready` debería ser `return (c == 'y')`.

Ahora podemos ver ejemplos de E/S más complicados. En primer lugar, la función `getLine`:

```
getLine     :: IO String
getLine     =  do c <- getChar
                  if c == '\n'
                       then return ""
                       else do l <- getLine
                               return (c:l)
```

Obsérvese el segundo `do` en la parte else. Cada `do` inicia una secuencia de sentencias. Cualquier otra construcción que forme parte de la función , como `if` en el ejemplo, debe usar un nuevo `do` para iniciar otras secuencias de acciones.

La función `return` da entrada a un valor en el dominio de las acciones de E/S. Pero, ¿qué pasa con la dirección inversa? ¿es posible invocar alguna acción de E/S desde una expresión normal ? Por ejemplo, ¿cómo podemos expresar `x + print y` en una expresión de modo que `y` sea imprimida al evaluar la expresión? La respuesta es que esto no es posible. No es posible adentrarse en el universo imperativo en mitad de código funcional puro. Cualquier valor `infectado' por acciones imperativas debe ser etiquetado como tal (usando el tipo `IO`). Una función como

```
f    ::  Int -> Int -> Int
```

no puede realizar ninguna E/S ya que `IO` no forma parte del tipo devuelto. Este hecho es habitualmente desconsolador para programadores acostumbrados a colocar sentencias tipo *print* liberalmente a lo largo del código de un programa durante la fase de depuración. Realmente, existen funciones peligrosas (rompen el carácter puro del lenguaje) para estos casos, aunque es mejor reservar el uso de éstas a programadores avanzados. Los paquetes de depuración suelen hacer un uso liberal de estas "funciones prohibidas" de un modo totalmente seguro.

7.2 Programando con acciones

Las acciones de E/S son valores normales en Haskell: pueden ser pasados como argumentos a funciones, pueden formar parte de estructuras, y en general, ser usados como cualquier otro valor. Considérese esta lista de acciones:

```
todoList :: [IO ()]

todoList = [putChar 'a',
            do putChar 'b'
               putChar 'c',
            do c <- getChar
               putChar c]
```

Esta lista no ejecuta ninguna acción - simplemente las almacena. Para enlazar estas acciones y dar lugar a una única acción, es necesaria una función como `sequence_`:

```
sequence_        :: [IO ()] -> IO ()
sequence_ []     =  return ()
sequence_ (a:as) =  do a
                       sequence_ as
```

Esta definción puede ser simplificada si observamos que la expresión `do x;y` es traducida a `x >> y`. Este patrón de recursión es el definido por la función `foldr`; una definición mejor de `sequence_` es:

```
sequence_        :: [IO ()] -> IO ()
sequence_        =  foldr (>>) (return ())
```

La notación `do` es útil pero en este caso el operador monádico subyacente , `>>`, es más apropiado. El conocimiento de los operadores utilizados para traducir la notación `do` puede ser muy útil para el programador.

La función `sequence_` puede ser usada para construir `putStr` a partir de `putChar`:

```
putStr                   :: String -> IO ()
putStr s                 =  sequence_ (map putChar s)
```

Una de las diferencias entre Haskell y la programación imperativa convencional puede verse en la definición de `putStr`. En un lenguaje imperativo, la aplicación de una versión imperativa de `putChar` sobre una cadena de caracteres daría lugar a que ésta se imprimiese. En Haskell, la función `map` no produce efecto alguno. Simplemente crea una lista de acciones, una por cada carácter en la cadena. La operación de plegado en la función `sequence` usa el operador `>>` para combinar todas las acciones individuales dando lugar a una acción única. La expresión `return ()` usada es totalmente necesaria -- `foldr` necesita una acción nula al final de la cadena de acciones que crea (¡sobre todo si la cadena de caracteres está vacía!).

El Prelude y las bibliotecas del lenguaje contienen varias funciones útiles para combinar acciones de E/S. La mayoría de estas funciones están genérizadas a mónadas arbitrarias; cualquier función que incluya el contexto `Monad m =>` en su tipo puede ser usada con el tipo `IO` (ya que éste es una instancia de la clase `Monad`).

7.3 Tratamiento de excepciones

Hasta ahora, hemos evitado el tratamiento de excepciones en las operaciones de E/S. Pero, ¿qué ocurriría si `getChar` se encontrase el final de un fichero? (Usaremos el término *error* para denotar el valor _|_: un error no recuperable como la no terminación o un error de patrones. Las excepciones, por otro lado, pueden ser capturadas y tratadas dentro de la mónada de E/S.) Un mecanismo de tratamiento, parecido al de Standard ML, es usado para tratar excepciones tales como "fichero no existente". No se usa ninguna sintaxis o semántica especial; el tratamiento de excepciones es parte de la definición de las operaciones de E/S.

Los errores son codificados usando un tipo de datos denominado `IOError`. Este tipo representa todas las posibles excepciones que pueden ocurrir al ejecutar operaciones de la mónada de E/S. El tipo es abstracto: los constructores no están disponibles para el usuario. Algunos predicados permiten inspeccionar datos del tipo `IOError`. Por ejemplo, la función

```
isEOFError       :: IOError -> Bool
```

determina si el error que toma como argumento fue causado por una condición de final de fichero. Al ser el tipo abstracto, los diseñadores del lenguaje pueden añadir nuevos tipos de errores al sistema sin notofocar el cambio en la implementación del tipo.

Un *manejador de excepciones* tiene como tipo `IOError -> IO a`. La función `catch` asocia un manejador de excepciones con una acción o un conjunto de éstas:

```
catch                    :: IO a -> (IOError -> IO a) -> IO a
```

Los argumentos de `catch` son una acción y el correspondiente manejador. Si la acción concluye sin error, el resultado de ésta es devuelto sin invocar al manejador. En otro caso, si se produce un error, éste es pasado al manejador como un valor de tipo `IOError` y la acción asociada con el manejador es invocada. Por ejemplo, la siguiente versión de `getChar` devuelve el carácter correspondiente al salto de línea cuando se produce un error:

```
getChar'                 :: IO Char
getChar'                 = getChar `catch` (\e -> return '\n')
```

Esto es bastante tosco ya que el manejador trata cualquier error del mismo modo. Si solo se quieren tratar los errores provocados por el final de un fichero, el error debe ser examinado:

```
getChar'           :: IO Char
getChar'           =  getChar `catch` eofHandler where
  eofHandler e = if isEofError e then return '\n' else ioError e
```

La función `ioError` eleva una excepción, que podrá ser tratada por otro manejador. El tipo de `ioError` es

```
ioError            :: IOError -> IO a
```

Es parecida a `return` pero hace que el control se transfiera al manejador de excepciones en vez de a la siguiente operación de E/S. El uso anidado de `catch` está permitido, y da lugar a manejadores de excepciones anidados.

Usando `getChar'`, podemos redefinir `getLine` como ejemplo del uso de manejadores anidados:

```
getLine'        :: IO String
getLine'        = catch getLine'' (\err -> return ("Error: " ++ show
err)) where
                   getLine'' = do c <- getChar'
                                  if c == '\n' then return ""
                                               else do l <- getLine'
                                                       return (c:l)
```

El manejador de excepciones anidado permite que `getChar'` trate los errores de fin de fichero, mientras que otros errores hacen que `getLine'` devuelva una cadena de caracteres comenzado por `"Error: "`.

Haskell define un manejador de excepciones por defecto en el nivel más externo de un programa cuyo comportamiento consiste en imprimir un mensaje con la excepción producida y finalizar el programa.

7.4 Ficheros, Canales y Manejadores

Aparte del uso de la mónada de E/S y del mecanismo de excepciones, el conjunto de operaciones de E/S proporcionadas por Haskell es muy parecido al de otros lenguajes. Muchas de estas operaciones están definidas en la biblioteca `IO` y por ello deben importarse explícitamente para ser usadas (los módulos y la importación de elementos son estudiados en la sección 11). Además, muchas de estas funciones son descritas en el informe de las bibliotecas del lenguaje en vez de en el informe del lenguaje.

La apertura de un fichero permite obtener un *manejador* (*handle*), con tipo `Handle`, que puede ser usado para operaciones de E/S. Al cerrar el manejador, se cierra el fichero asociado:

```
type FilePath         =  String  -- nombres de ficheros
openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()
data IOMode           =  ReadMode | WriteMode | AppendMode |
ReadWriteMode
```

Los manejadores también pueden ser asociados con canales (*channels*): puertos de comunicación que no están conectados directamente con ficheros. Algunos manejadores de ficheros están predefinidos, como por ejemplo `stdin` (la entrada estándar), `stdout` (la salida estándar), y `stderr` (el canal de errores estándar). Dos operaciones de E/S a nivel de caracteres son `hGetChar` y `hPutChar`, que toman un manejador como argumento. La función `getChar` usada previamente puede ser definida del siguiente modo:

```
getChar                = hGetChar stdin
```

Haskell también permite leer el contenido completo de un fichero o canal como una cadena de caracteres:

```
getContents            :: Handle -> String
```

Puede parecer que `getContents` debe leer inmediatamente todo el contenido del fichero o canal, dando lugar a un rendimiento pobre desde el punto de vista del espacio de memoria y tiempo utilizados. Sin embargo, esto no es lo que ocurre. Esto se debe a que `getContents` devuelve una lista de caracteres "perezosa" (recuérdese que las cadenas de caracteres son listas de caracteres en Haskell), cuyos elementos son leídos del fichero "bajo demanda" (del mismo modo que son generadas las listas normales). Puede esperarse que las implementaciones de esta función lean uno por uno los caracteres del fichero según son necesitados para proseguir el cómputo realizado con la cadena.

En el siguiente ejemplo, se copia el contenido de un fichero en otro:

```
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
```

```
          contents   <- getContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."

getAndOpenFile          :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile mode name)
             (\_ -> do putStrLn "(Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

Al usar la función perezosa `getContents`, no es necesario leer todo el contenido del fichero en memoria de una vez. Si `hPutStr` utilizase un *buffer* para escribir la cadena en bloques de tamaño fijo, solo uno de estos bloques de caracteres ha de permanecer en memoria simultáneamente. El fichero de entrada es cerrado de modo implícito cuando el último carácter es leído.

7.5 Haskell y la programación imperativa

La programación de operaciones de E/S puede dar lugar a la siguiente reflexión: el estilo utilizado se parece sospechosamente al de la programación imperativa. Por ejemplo, la función `getLine`:

```
getLine         = do c <- getChar
                     if c == '\n'
                          then return ""
                          else do l <- getLine
                                  return (c:l)
```

guarda una similitud estrecha con el siguiente código imperativo (que no está escrito en ningún lenguaje concreto) :

```
function getLine() {
  c := getChar();
  if c == `\n` then return ""
               else {l := getLine();
                     return c:l}}
```

Así, después de todo, ¿ ha reinventado Haskell simplemente la rueda imperativa ?

En cierto sentido, la respuesta es sí. La mónada de E/S constituye un pequeño sub-lenguaje imperativo dentro de Haskell, de modo que la componente de E/S de un programa puede parecer similar al código de un lenguaje imperativo habitual. Pero existe una diferencia importante: no es necesaria una semántica especial para ello. Concretamente, el razonamiento ecuacional no deja de ser válido. La apariencia imperativa del código monádico no denigra el aspecto funcional de Haskell. Un programador funcional experto debe ser capaz de minimizar la componente imperativa de un programa, usando únicamente la mónada de E/S en una parte mínima del programa. La mónada claramente separa las componentes imperativa y funcional del programa. Por el contrario, los lenguajes imperativos con subconjuntos funcionales no establecen una barrera bien definida entre la parte funcional pura y la parte imperativa del programa.

8 Las clases estandarizadas de Haskell

En esta sección, introduciremos las clases estándar predefinidas de Haskell. Hemos simplificado algo las clases omitiendo algunos de los métodos menos interesantes; el informe de Haskell contiene una descripción más completa. Por último, algunas de las clases estándar son parte de las bibliotecas de Haskell; puede verse una descripción de éstas en el informe de bibliotecas de Haskell.

8.1 Las clases con igualdad y orden

Las clases `Eq` y `Ord` han sido ya discutidas. La definición de `Ord` en el Prelude es más compleja que la versión simplificada presentada antes. En particular, observe el método `compare`:

```
data Ordering            = EQ | LT | GT
compare                  :: Ord a => a -> a -> Ordering
```

El método `compare` es suficiente para que queden definidos todos los otros métodos (por defecto) en esta clase y es la mejor forma de crear instancias de `Ord`.

8.2 La clase Enum

La clase `Enum` posee un conjunto de operaciones que subyacen bajo la sintaxis de las secuencias aritméticas; por ejemplo, la secuencia `[1,3..]` es simplemente una sintaxis más cómoda para la expresión `enumFromThen 1 3` (véase §3.10 para la traducción formal). Podemos observar ahora que es posible usar las secuencias aritméticas para generar listas de cualquier tipo que sea una instancia de la clase `Enum`. Esto incluye no solo la mayoría de los tipos numéricos, sino que también al tipo `Char`, de modo que, por ejemplo, `['a'..'z']` denota la lista de todas las letras minúsculas en orden alfabético. Además, es fácil realizar la correspondiente instancia de la clase `Enum` para los tipos enumerados definidos por el programador, como `Color`. Si se ha realizado ésta, entonces tendremos:

```
[Red..Violet] => [Red, Green, Blue, Indigo, Violet]
```

Obsérvese que la secuencia es aritmética en el sentido de que el incremento entre valores es constante, aunque los valores no son números. La mayoría de los tipos instancias de `Enum` pueden ser transladados a enteros; para estos tipos, las funciones `fromEnum` y `toEnum` convierten entre `Int` y un tipo en la clase `Enum`.

8.3 Las clases Read y Show

Las instancias de la clase `Show` son aquellos tipos que pueden ser convertidos en cadenas de caracteres (habitualmente para realizar operaciones de E/S). La clase `Read` proporciona operaciones para analizar (*parse*) cadenas de caracteres y obtener los valores que éstas representan. La función más simple de la clase `Show` es `show`:

```
show                     :: (Show a) => a -> String
```

Como puede esperarse, `show` toma cualquier valor del tipo apropiado y devuelve su representación como una cadena de caracteres (una lista de caracteres); por ejemplo, `show (2+2)`, produce como resultado `"4"`. Sin embargo, habitualmente necesitaremos producir cadenas de caracteres más complejas partir de varios valores, como en el caso de

```
"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++
"."
```

pero todas estas concatenaciones resultan un poco ineficientes. Concretamente, consideremos una función para representar los árboles binarios de la sección 2.2.1 como cadenas de caracteres, con marcadores adecuados para reflejar el anidamiento de los subárboles y la distinción entre las ramas izquierda y derecha (supuesto que el tipo de los elementos almacenados en el árbol es representable como una cadena de caracteres):

```
showTree                :: (Show a) => Tree a -> String
showTree (Leaf x)       =  show x
showTree (Branch l r)   =  "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Dado que `(++)` tiene complejidad lineal sobre la longitud de su argumento izquierdo, `showTree` tiene complejidad cuadrática sobre el tamaño del árbol.

El método `shows` puede ser usado para obtener una complejidad lineal:

```
shows                   :: (Show a) => a -> String -> String
```

`shows` toma un valor mostrable y una cadena, y devuelve la concatenación de la representación del valor con la cadena segundo argumento. El segundo argumento hace las veces de una cadena acumuladora, y `show` puede ser definido como `shows` con la cadena vacía como acumulador. Aquí aparece la definición por defecto de la función `show` en la clase `Show`:

```
show x                  =  shows x ""
```

Podemos usar `shows` para definir una versión más eficiente de `showTree`, que también usa un argumento como cadena acumuladora:

```
showsTree               :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s    =  shows x s
showsTree (Branch l r) s=  '<' : showsTree l ('|' : showsTree r ('>' :
s))
```

Esto resuelve el problema de eficiencia original (`showsTree` tiene complejidad lineal), aunque la representación de esta función (y de otras similares) puede ser mejorada. Comencemos por definir un sinónimo de tipo:

```
type ShowS              =  String -> String
```

Este es el tipo de una función que devuelve la representación como cadena de caracteres de algo seguida de una cadena acumuladora. En segundo lugar, podemos evitar trabajar explícitamente con acumuladores, y a la vez, evitar el apilamiento de paréntesis en la parte derecha de la definición, si usamos la composición de funciones:

```
showsTree               :: (Show a) => Tree a -> ShowS
showsTree (Leaf x)      =  shows x
showsTree (Branch l r)  =  ('<':) . showsTree l . ('|':) . showsTree r .
('>':)
```

Con esto hemos hecho algo más que ordenar un poco el código: hemos pasado de una representación a *nivel de objetos* (en este caso, cadenas de caracteres) a una representación a *nivel de funciones*. Podemos considerar que el tipo de la función indica que `showsTree` convierte un árbol en una *función mostradora*. Las funciones como `('<' :)` o `("a string" ++)` son funciones mostradoras primitivas, y podemos construir funciones más complejas usando la composición de funciones.

Ya que podemos transformar árboles en cadenas de caracteres, consideremos el problema inverso. La idea básica es construir un analizador (*parser*) para un tipo `a`, como una función que toma una cadena de caracteres y devuelve una lista de pares con tipo `(a, String)`[9]. El Prelude define un sinónimo de tipo para estas funciones:

```
type ReadS a            =  String -> [(a,String)]
```

Normalmente, un analizador devuelve una lista con un solo elemento, conteniendo el valor de tipo `a` que fue leído de la cadena de entrada y el resto de la cadena que queda por analizar. Si el análisis no fue posible, entonces, el resultado es la lista vacía, y si hay más de un análisis posible (ambiguedad), la lista resultante contendrá más de un par. La función `reads` se comporta como un analizador para cualquier tipo instancia de la clase `Read`:

```
reads                   :: (Read a) => ReadS a
```

Podemos usar esta función para definir una función analizadora para la representación mediante cadenas de caracteres de los árboles binarios producida por `showsTree`. Las listas por comprensión son un mecanismo adecuado para construir tales analizadores (Una aproximación aún más elegante es usar mónadas y combinadores de analizadores. Existe una biblioteca para esto que es distribuida con la mayoría de los sistemas Haskell):

```
readsTree               :: (Read a) => ReadS (Tree a)
readsTree ('<':s)       =  [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                              (r, '>':u) <- readsTree t ]
readsTree s             =  [(Leaf x, t)     | (x,t)      <- reads s]
```

Examinemos esta función en detalle por un momento. Los casos a considerar son dos: Si el primer carácter de la cadena a analizar es `'<'`, deberíamos tener la representación de una rama; en otro caso, tenemos una hoja. En el primer caso, si denotamos al resto de la cadena de entrada a partir del caracter menor (`'<'`) con el nombre `s`, cualquier análisis posible debe ser un árbol con la forma `Branch l r` donde el resto de la cadena por analizar puede ser llamado `u`, siempre y cuando:

1. El árbol `l` pueda ser obtenido analizando la cadena `s` desde el inicio.
2. El resto de la cadena (después de la representación de `l`) debe comenzar con el caracter `'|'`. Llamemos a la cola de esta cadena `t`.
3. El árbol `r` puede ser analizado a partir del comienzo de `t`.
4. La cadena sobrante de dicho análisis debe comenzar por `'>'`, y u debe ser su cola.

Obsérvese la capacidad expresiva que se obtiene de la combinación del uso de patrones y las listas por comprensión: La forma del análisis resultante viene dado por la expresión principal de la lista por comprensión, las dos primeras condiciones anteriores vienen reflejadas por el hecho de que el primer generador ("`(l, '|':t)` usa la lista de análisis de `s`."), y las demás condiciones quedan reflejadas con el segundo generador.

La segunda ecuación de la definición anterior dice simplemente que para analizar la representación de una hoja, es necesario analizar un elemento del árbol y aplicar el constructor `Leaf` al valor obtenido.

Aceptaremos por el momento que existe una instancia de la clase `Read` (y `Show`) para el tipo `Integer` (y para otros más), que proporciona una función `reads` que se comporta como es de esperar, por ejemplo:

```
(reads "5 golden rings") :: [(Integer,String)] => [(5, " golden rings")]
```

A partir de esto, el lector debería verificar las siguientes evaluaciones:

```
readsTree "<1|<2|3>>"      => [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]

readsTree "<1|2"           => []
```

Hay un par de limitaciones en la definición dada de `readsTree`. Una es que el analizador es demasiado rígido, y no permite espacios en blanco antes o entre los elementos que forman parte de la representación del árbol; la otra es que el modo en el que se analizan los delimitadores es bastante distinto del modo en que se analizan los valores hoja y los subárboles, lo cual no es uniforme y hace que la definición sea más difícil de comprender. Podemos solucionar ambos problemas usando el analizador léxico que se define en el Prelude:

```
lex                     :: ReadS String
```

`lex` devuelve normalmente una lista con un solo elemento que es un par de cadenas de caracteres: el primer lexema en la cadena de entrada y el resto de la cadena de entrada. La reglas léxicas son las de Haskell, incluyendo el tratamiento de comentarios, que `lex` ignora, junto con los espacios en blanco. Si la cadena de entrada está vacía o contiene tan solo espacios en blanco y comentarios, `lex` devuelve `[("","")]`; si la entrada no está vacía, pero tampoco comienza con un lexema válido tras los espacios en blanco y comentarios iniciales, `lex` devuelve `[]`.

Usando el analizador léxico, el analizador para el árbol puede ser definido del siguiente modo:

```
readsTree               :: (Read a) => ReadS (Tree a)
readsTree s             =  [(Branch l r, x) | ("<", t) <- lex s,
                                              (l,   u) <- readsTree t,
                                              ("|", v) <- lex u,
                                              (r,   w) <- readsTree v,
                                              (">", x) <- lex w        ]
```

```
                        ++
                        [(Leaf x, t)     | (x,   t) <- reads s       ]
```

Podemos usar `readsTree` y `showsTree` para hacer que el tipo`(Read a) => Tree a` sea una instancia de la clase `Read` y `(Show a) => Tree a` una instancia de `Show`. Esto nos permitirá usar las funciones sobrecargadas genéricas del Prelude para analizar y mostrar árboles. Además, será posible automáticamente analizar y mostrar otros tipos que contengan árboles como componentes, por ejemplo, `[Tree Integer]`. Tal como están definidos, `readsTree` y `showsTree` tienen prácticamente los tipos adecuados para ser métodos de las clases `Show` y `Read`. Los métodos `showPrec` y `readPrec` son versiones parametrizadas de `shows` y `reads`. EL parámetro extra es un nivel de precedencia, usado para colocar paréntesis de forma adecuada cuando se utilizan constructores infijos. Para tipos tales como `Tree`, la precedencia es ignorada. Las instancias de `Show` y `Read` para `Tree` son:

```
instance Show a => Show (Tree a) where
      showPrec _ x = showsTree x

instance Read a => Read (Tree a) where
      readsPrec _ s = readsTree s
```

Esto, sin embargo, es una versión muy ineficiente de Show. Observe que la clase `Show` define por defecto los métodos `showsPrec` y `show`, permitiendo al usuario definir uno de ellos en una declaración de instancia. Dado que cada uno se define en función del otro, una declaración de instancia que no defina ninguno entrará en un bucle indefinido cuando se llame a uno de ellos. Otras clases tales como `Num` también tienen estos "interbloqueos por defecto".

El lector interesado en profundizar sobre `Read` y `Show` puede consultar §D para más detalles.

Podemos probar las instancias de `Read` y `Show` aplicando la siguiente composición de funciones `(read . show)` (que debe comportarse como la función identidad) a algunos árboles, donde `read` es una especialización de `reads`:

```
read                    :: (Read a) => String -> a
```

Esta función produce un error si no hay un análisis único o si la entrada contiene algo más que la representación de un valor del tipo `a` y comentarios y espacios en blanco opcionales.

8.4 Instancias derivadas

Recuerde la instancia `Eq` para árboles que presentamos en la Sección 5; tal declaración es simple---y laboriosa--- de construir: requerimos que el tipo de los elementos de las hojas dispongan de la igualdad; entonces, dos hojas son iguales si y sólo si contienen elementos iguales, y dos ramas son iguales si y sólo si sus subárboles izquierdo y derecho son iguales respectivamente. Cualesquiera otros dos árboles son distintos:

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)     == (Leaf y)        =  x == y
```

```
    (Branch l r) == (Branch l' r')  =  l == l' && r == r'
    _            == _               =  False
```

Afortunadamente, no necesitamos realizar esta tediosa tarea cada vez que necesitemos el operador de igualdad para un nuevo tipo; la instancia `Eq` puede ser *derivada automáticamente* desde la propia declaración:

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

La cláusula `deriving` produce de forma implícita una declaración de instancia `Eq` justo igual que la de la Sección 5. Las instancias de `Ord`, `Enum`, `Ix`, `Read`, y `Show` pueden ser generadas por la cláusula `deriving`. [Tras `deriving`, más de una clase puede ser especificada, en cuyo caso la lista de nombres deberá ir entre paréntesis y separados por comas.]

La instancia `Tree` para la clase `Ord` es ligeramente más complicada que la instancia `Eq`:

```
instance  (Ord a) => Ord (Tree a)  where
    (Leaf _)     <= (Branch _)      =  True
    (Leaf x)     <= (Leaf y)        =  x <= y
    (Branch _)   <= (Leaf _)        =  False
    (Branch l r) <= (Branch l' r')  =  l == l' && r <= r' || l <= l'
```

Esto especifica un orden *lexicográfico*: los constructores están ordenados según el orden en que aparecen en la declaración `data`, y los argumentos de un constructor son comparados de izquierda a derecha. Recuerde que el constructor de listas es semánticamente equivalente a un constructor ordinario de dos argumentos. En efecto, la siguiente es su declaración completa:

```
data [a]        = [] | a : [a] deriving (Eq, Ord)     -- pseudo-código
```

(Las listas son también instancias de `Show` y `Read`, pero estas no aparecen como derivadas.) Las instancias derivadas de `Eq` y `Ord` para las listas son las usuales; en particular, las cadenas, como listas de caracteres, están ordenadas según determina el tipo subyacente `Char`, donde una subcadena inicial de otra es considerada menor; por ejemplo, `"cat" < "catalog"`.

En la práctica en la mayoría de las ocasiones, las instancias de `Eq` y `Ord` son derivadas en lugar de definidas por el usuario. Sólo deberíamos dar las definiciones de igualdad y ordenación cuando éstas sean especiales, siendo cuidadosos en mantener las propiedades algebraicas de relación de equivalencia y orden total respectivamente que se esperan de estas relaciones. Por ejemplo, un predicado `(==)` no transitivo, puede ser desastroso, confundiendo a los lectores del programa y confundiendo a cualquier transformación de programas, ya sea manual o automática que confía en que el predicado `(==)` sea una aproximación de la definición de igualdad. No obstante, a veces es necesario proveer instancias de `Eq` u `Ord` diferentes de aquellas que serían derivadas; probablemente el ejemplo más significativo sea el de un tipo abstracto de datos en el cual dos valores concretos diferentes puedan representar el mismo valor abstracto.

Un tipo enumerado puede tener una instancia derivada de `Enum`, y aquí de nuevo, el orden es el de los constructores en la declaración `data`. Por ejemplo:

```
data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday         deriving (Enum)
```

Aquí hay algunos ejemplos simples del uso de la instancia derivada para este tipo:

```
[Wednesday..Friday]    => [Wednesday, Thursday, Friday]

[Monday, Wednesday ..] => [Monday, Wednesday, Friday]
```

Es posible realizar instancias derivadas de `Read` (`Show`) para aquellos tipos cuyas componentes también sean derivadas de `Read` (`Show`). (Las instancias de `Read` y `Show` para la mayoría de los tipos estándar están disponibles en el Prelude. Algunos tipos, tales como el tipo función `(->)`, tienen una instancia `Show` pero no su correspondiente `Read`.) La representación textual definida por una instancia derivada `Show` es consistente con la apariencia de las expresiones constantes en Haskell del tipo en cuestión. Por ejemplo, si añadimos `Show` y `Read` a la cláusula `deriving` para el tipo `Day`, anterior, obtenemos

```
show [Monday..Wednesday] => "[Monday,Tuesday,Wednesday]"
```

9 Sobre las Mónadas

La mayoría de los recien llegados a Haskell quedan desconcertados ante el concepto de *mónada*. Las mónadas aparecen a menudo en Haskell: el sistema de E/S está diseñado usando una mónada, se proporciona una sintaxis especial para el uso de las mónadas (las expresiones do), y uno de los módulos en las librerías estandarizadas está dedicado por completo a las mónadas. En esta sección exploramos la programación monádica con mayor detalle.

Esta sección del tutorial es posiblemente menos `agradable' que las otras. Aquí trataremos no solo las características del lenguaje relacionadas con las mónadas sino que también intentaremos dar una visión más amplia: mostraremos porque son las mónadas una herramienta tan importante y como son usadas. No hay un módo único de explicar las mónadas que sea adecuado para todo el mundo; otras explicaciones encontrarse en `haskell.org`. Una buena introducción al uso práctico de las mónadas es el artículo Monads for Functional Programming [10] de Wadler.

9.1 Las clases monádicas

El Prelude contiene varias clases que definen mónadas del modo en que son usadas en Haskell. Estas clases están basadas en el concepto de mónada que aparece en la teoría de categorías; aunque la terminología en esta teórica proporciona los nombres para las clases monádicas y las correspondientes operaciones, no es necesario ahondar en las matemáticas abstractas para obtener una idea intuitiva de cómo usar las clases monádicas..

Una mónada se construye sobre un tipo polimórfico como `IO`. La mónada propiamente queda definida mediante declaraciones de instancia que asocian el tipo con algunas o todas las clases monádicas, `Functor`, `Monad`, y `MonadPlus`. Ninguna de las clases monádicas puede ser derivada. Además de `IO`, otros dos tipos en el Prelude son miembros de las clases monádicas: las listas (`[]`) y el tipo `Maybe`.

Matemáticamente, las mónadas están caracterizadas por un conjunto de *leyes* (o propiedades) que deberían cumplir las operaciones monádicas. Este concepto de ley no es exclusivo de las mónadas: otras de las operaciones de Haskell están caracterizadas, al menos informalmente, por leyes. Por ejemplo, `x /= y` y `not (x == y)` deberían ser lo mismo para cualquier tipo de valor para el que tenga sentido dicha comparación. Sin embargo, no hay garantía de de que esto ocurra: tanto `==` como `/=` son métodos independientes en la la clase `Eq` y no hay modo de asegurar que == y =/ estén relacionados de este modo. Del mismo modo, las leyes monádicas que presentamos a continuación no son impuestas por Haskell, pero deberían cumplirse para cualquier instancia de una clase monádica. Las leyes de las mónadas proporcionan un mayor entendimiento de la estructura subyacente de las mónadas: esperamos que al examinar estas leyes proporcionemos al lector una primera impresión de cómo se usan las mónadas.

La clase `Functor`, ya discutida en la sección 5, define una única operación: `fmap`. La función fmap aplica una operación a los objetos pertenecientes a un contenedor (los tipos polimórficos pueden ser considerados como contenedores de valores de otro tipo), devolviendo un contenedor con la misma forma. Estas leyes caracterizan a `fmap` en la clase `Functor`:

```
fmap id      = id

fmap (f . g) = fmap f . fmap g
```

Estas leyes aseguran que la forma del contenedor no es modificada por `fmap` y que el contenido de un contenedor no es reorganizado por la aplicación.

La clase `Monad` define dos operaciones básicas: `>>=` (*bind*) y `return`.

```
infixl 1  >>, >>=
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a

    m >> k           =  m >>= \_ -> k
```

Las operaciones `>>` y `>>=`, combinan dos valores monádicos mientras que la función `return` sumerje un valor en la mónada (el contenedor). El tipo de `>>=`, `Monad m => m a -> (a -> m b) -> m b`, nos ayuda a comprender esta operación: `ma >>= \v -> mb` combina un valor monádico `ma` que contenga valores de tipo `a` y una función que opera sobre sobre un valor `v` de tipo `a` devolviendo el valor monádico `mb`. El resultado es combinar `ma` y `mb` en un valor monádico que contenga `b`. El operador `>>` es usado cuando la función no usa el valor producido por la primera operación monádica.

El significado de los operadores *bind* depende, por supesto, de la mónada. Por ejemplo, en la mónada IO (mónada de E/S), `x >>= y` lleva a cabo dos acciones secuencialmente, pasando el resultado de la primera a la segunda. Para las otras mónadas predefinidas, las listas y el tipo `Maybe`, estas operaciones monádicas pueden ser entendidas en términos de pasar cero o más valores desde una computación a la próxima. Veremos ejemplos de esto a continuación.

La notación `do` proporciona una abreviatura sintáctica simple para encadenamientos de operaciones monádicas. La esencia de traducción de las expresiones `do` está reflejada en las siguientes dos reglas:

```
  do e1 ; e2      =        e1 >> e2
  do p <- e1; e2  =        e1 >>= \p -> e2
```

Cuando el patrón en la segunda ecuación `do` es refutable, un fallo en el encaje de patrones llama a la función `fail`. Esto puede elevar un error (como ocurre en la mónada `IO`) o devolver un `cero' (como ocurre en la mónada lista). Así, la traducción completa es

```
  do p <- e1; e2  =   e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

donde `"s"` es una cadena de caracteres identificando la localización de la sentencia `do` para un posible uso en un mensaje de error. Por ejemplo, en la mónada IO, una acción como `'a' <- getChar` llamará a `fail` si el caracter tecleado no es 'a'. Esto, a su vez, terminará el programa ya que en la mónada IO `fail` llama a `error`.

Las leyes que caracterizan a >>= y return son:

```
return a >>= k          = k a

m >>= return            = m

xs >>= return . f       = fmap f xs

m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

La clase MonadZero se usa para mónadas que definen un elemento *cero* y una operación *plus*:

```
class  (Monad m) => MonadZero m  where
    mzero             :: m a
    mplus             :: m a -> m a -> m a
```

El elemento cero debe verificar las siguientes propiedades:

```
m >>= \x -> mzero = mzero

mzero >>= m       = mzero
```

Para las listas, el valor cero es [], la lista vacía. La mónada IO no posee un cero por lo que no es un miembro de esta clase.

Las propiedades que caracterizan al operador mplus son las siguientes:

```
m `mplus` mzero = m

mzero `mplus` m = m
```

En el caso de las listas, el operador mplus es la concatenación habitual.

9.2 Mónadas predefinidas

A partir de las operaciones monádicas y las correspondientes propiedades, ¿Qué podemos hacer? Dado que ya hemos examinado la mónada IO detalladamente, comenzaremos por las otras dos mónadas predefinidas.

Para las listas, el operador de *binding* se corresponde con agrupar un conjunto de computaciones para cada valor de la lista. Cuando se usa sobre listas, el tipo de >>= es:

```
(>>=)                    :: [a] -> (a -> [b]) -> [b]
```

Esto se interpreta como, dada una lista cuyos elementos tienen tipo a y una función que transforma un valor de tipo a en una lista de valores de tipo b, el operador de *binding* aplica esta función a cada uno de los valores de tipo a en la entrada y devuelve todos los

valores de tipo `b` generados como una lista. La función `return` crea una lista unitaria. Estas operaciones deberían ser ya familiares: las listas por comprensión pueden ser expresadas fácilmente usando las operaciones monádicas definidas para las listas. Las siguientes tres expresiones corresponden a diferentes modos de escribir lo mismo:

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

La definición depende del hecho de que la definición de la función `fail` para la mónada lista es la lista vacía. Esencialmente, cada `<-` genera un conjunto de valores que se pasa al resto de la computación monádica. Así `x <- [1,2,3]` invoca el resto de la computación monádica tres veces, una por cada elemento en la lista. La expresión devuelta, `(x,y)`, será evaluada para todas las posibles combinaciones de valores. En este sentido, podemos ver que la mónada lista es útil para funciones que trabajan con argumentos multi-valuados. Por ejemplo, la siguiente función:

```
mvLift2                :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y          =  do  x' <- x
                              y' <- y
                              return (f x' y')
```

convierte una función ordinaria de dos argumentos (`f`) en una función sobre múltiples valores (listas de argumentos), devolviendo un valor para cada posible combinación de dos argumentos de entrada. Por ejemplo,

```
mvLift2 (+) [1,3] [10,20,30]    => [11,21,31,13,23,33]

mvLift2 (\a b->[a,b]) "ab" "cd" => ["ac","ad","bc","bd"]

mvLift2 (*) [1,2,4] []          => []
```

La función es una especialización de la función `liftM2` definida en la biblioteca estandarizada para mónadas. Puedes verla como una función que transporta a cierta función, `f`, desde fuera de la mónada de las listas a la mónada de las listas en la que los cómputos se realizan sobre valores múltiples.

La mónada definida para el tipo `Maybe` es parecida a la mónada de las listas: el valor `Nothing` hace el papel de `[]` y `Just x` el de `[x]`.

9.3 Usando Mónadas

Explicar tan solo los operadores monádicos y las propiedades asociadas a estos no es suficiente para mostrar la utilidad de las mónadas. Lo que las mónadas realmente proporcionan es *modularidad*. Con esto queremos decir que, al definir una operación de modo monádico,

podemos ocultar la maquinaria subyacente de modo que es posible añadir características a la mónada de un modo transparente. El artículo [10] de Wadler es un ejemplo excelente de cómo pueden usarse las mónadas para construir programas modulares. Comenzaremos con una mónada tomada directamente de dicho artículo, la mónada de los estados, para construir posteriormente una mónada más compleja con una definición similar.

De un modo breve, una mónada de estados construída sobre un tipo `S` toma la siguiente forma:

```
data SM a = SM (S -> (a,S))  -- The monadic type

instance Monad SM where
  -- define la propagación de estados
  SM c1 >>= fc2         =  SM (\s0 -> let (r,s1) = c1 s0
                                          SM c2 = fc2 r in
                                        c2 s1)
  return k              =  SM (\s -> (k,s))

 -- extrae el estado de la mónada
readSM                  :: SM S
readSM                  =  SM (\s -> (s,s))

 -- modifica el estado
updateSM                :: (S -> S) -> SM ()
updateSM f              =  SM (\s -> ((), f s))

-- ejecuta una computación en la mónada
runSM                   :: S -> SM a -> (a,S)
runSM s0 (SM c)         =  c s0
```

El ejemplo define un nuevo tipo, `SM`, para representar cómputos que implícitamente arrastran un valor de tipo `S`. Es decir, una computación de tipo `SM t` define un valor de tipo `t` que a la vez interactúa (leyendo y modificando) con un estado de tipo `S`. La definición `SM` es simple: se trata de funciones que toman un estado y producen dos resultados: el valor devuelto (de cualquier tipo) y el valor del estado actualizado. No podemos usar aquí un sinónimo de tipos: es necesario un nombre de tipo como `SM` para poder usarlo en declaraciones de instancia. En este caso, se suele usar una declaración `newtype` en vez de la declaración `data`.

La declaración de instancia define la `fontanería' correspondiente a la mónada: como secuenciar dos computaciones y la definición de la computación vacía. La secuenciación (el operador `>>=`) define un cómputo (denotado con el constructor `SM`) que pasa un estado inicial, `s0`, a `c1`, y entonces pasa el valor resultante de este cómputo, `r`, a la función que devuelve el segundo cómputo, `c2`. Por último, el estado resultado de `c1` se pasa a `c2` y el resultado final se define como el resultado de `c2`.

La defición de `return` es más simple: `return` no modifica el estado en absoluto; simplemente vale para convertir un valor en un cómputo monádico.

Aunque `>>=` y `return` son los operadores de secuenciación básicos para la mónada, tambien necesitamos algunas *primitivas monádicas*. Una primitiva monádica es

simplemente una operación que usa la representación interna de la mónada accediendo a las `ruedas y engranajes' que la hacen funcionar. Por ejemplo, en la mónada `IO`, operaciones como `putChar` son primitivas ya que se ocupan del funcionamiento interno de la mónada `IO`. De un modo similar, nuestra mónada de estados usa dos primitivas: `readSM` y `updateSM`. Obsérvese que estas dependen de la estructura interna de la mónada - un cambio en la definición del tipo `SM` requeriría que se realizase un cambio en estas primitivas.

Las definiciones de `readSM` y `updateSM` son simples: `readSM` presenta, con objeto de que sea observado, el estado de la mónada mientras que `updateSM` permite al usuario alterar el estado de la mónada. (Podríamos también haber usado una función como `writeSM` como primitiva pero la actualización es a menudo un modo más natural de manejar el estado).

Por último, necesitamos una función para ejecutar cómputos en la mónada, `runSM`. Ésta toma un valor inicial para el estado y un cómputo y devuelve tanto el valor del resultado del cómputo como el estado final.

De un modo más genérico, lo que estamos intentando es definir toda una computación como una serie de pasos (funciones con tipo `SM a`), secuenciadas mediante `>>=` y `return`. Estos pasos pueden interactuar con el estado (a través de `readSM` o `updateSM`) o pueden incluso ignorar el estado. En cualquier caso, el uso (o no uso) del estado se encuentra oculto: no invocamos o secuenciamos los cómputos de un modo diferente dependiendo de que usen o no `S`.

En lugar de presentar algún ejemplo usando esta mónada tan simple, avanzamos hacia un ejemplo más elaborado que engloba también la mónada de estados. Definimos un pequeño *lenguaje embebido* para el cálculo del uso de recursos. Con esto queremos decir que construiremos un lenguaje de propósito especial, implementándolo como un conjunto de tipos y funciones en Haskell, para construir una biblioteca de operaciones y tipos hechos a la medida del dominio de interés.

En este ejemplo, consideramos un cómputo que requiere alguna clase de recurso. Si el recurso está disponible, el cómputo prosigue; cuando el recurso no está disponible, el cómputo queda suspendido. Usaremos el tipo `R` para denotar una computación que use los recursos controlados por nuestra mónada. La definición de `R` es la siguiente:

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

Cada cómputo es una función de los recursos disponibles a los recursos sobrantes, emparejados con, o bien un resultado, de tipo `a`, o un cómputo suspendido, de tipo `R a`, que captura el trabajo hecho hasta el punto en el cual los recursos se agotaron.

La instancia para la clase `Monad` correspondiente al tipo `R` es:

```
instance Monad R where
  R c1 >>= fc2          = R (\r -> case c1 r of
                                (r', Left v)    -> let R c2 = fc2 v in
                                                   c2 r'
                                (r', Right pc1) -
> (r', Right (pc1 >>= fc2)))
  return v              = R (\r -> (r, (Left v)))
```

El tipo `Resource` se usa del mismo modo que el estado en la mónada de estados. Esta definición se puede interpretar del siguiente modo: para combinar dos cómputos `dependientes de recursos', `c1` y `fc2` (una función que produzca `c2`), pasamos los recursos iniciales a `c1`. El resultado será o bien

- un valor, `v`, y los recursos no utilizados, que serán utilizados para resolver el próximo cómputo (la llamada `fc2 v`), o
- un cómputo suspendido, `pc1`, y los recursos no utilizados en el momento de la suspensión.

La suspensión debe tener en cuenta el segundo cómputo: `pc1` suspende solamente el primer cómputo, `c1`, por lo que debemos seguir éste de `c2` para capturar la suspensión de todo el cómputo. La definición de `return` no cambia los recursos y a la vez introduce `v` en la mónada.

Esta declaración de instancia define la estructura básica de la mónada pero no indica cómo se usan los recursos. Así, esta mónada podría ser usada para manejar distintos tipos de recursos o para implementar distintas políticas para el uso de recursos. Como ejemplo, mostraremos una definición de recursos muy simple: haremos que el tipo `Resource` sea un `Integer`, que represente los pasos computacionales disponibles:

```
type Resource           =  Integer
```

Esta función consume un paso a no ser que no queden pasos disponibles:

```
step                    :: a -> R a
step v                  =  c where
                              c = R (\r -> if r /= 0 then (r-1, Left v)
                                                     else (r, Right c))
```

Los constructores `Left` y `Right` son parte del tipo `Either`. La función continúa con el cómputo en `R` y devuelve `v` siempre que exista al menos un paso computacional disponible. Si no hay pasos disponibles, la función `step` suspende el computo actual (esta suspensión queda capturada en `c`) y vuelve a introducir este cómputo suspendido en la mónada.

Hasta ahora, tenemos las herramientas necesarias para definir una secuencia de computaciones `dependientes de recursos' (la mónada) y podemos expresar una forma de uso de recursos mediante `step`. Por último, necesitamos considerar cómo se expresarán los cómputos correspondientes a esta mónada.

Consideremos una función de incremento para nuestra mónada:

```
inc                     :: R Integer -> R Integer
inc i                   =  do iValue <- i
                              step (iValue+1)
```

De este modo definimos el incremento como un único paso computacional. La `<-` es

necesaria para extraer el valor de argumento de la mónada; el tipo de `iValue` es `Integer` en vez de `R Integer`.

Esta definición no es totalmente satisfactoria, especialmente, si la comparamos con la definición estandarizada de la función de incremento. ¿Podríamos mejor `arropar' operaciones existentes como `+` para que funcionen en nuestro mundo monádico? Comenzaremos con un conjunto de funciones de `elevamiento` (*lifting*). Éstas permiten introducir funcionalidad existente en la mónada. Consideremos la definición de `lift1` (se trata de una definición ligeramente diferente a la función `liftM` de la biblioteca `Monad`):

```
lift1                   :: (a -> b) -> (R a -> R b)
lift1 f                 =  \ra1 -> do a1 <- ra1
                                      step (f a1)
```

Esta función toma una función de un único argumento, `f`, y crea una función `R` que ejecuta la función elevada en un único paso. Usando `lift1`, `inc` puede ser definida como

```
inc                     :: R Integer -> R Integer
inc                     =  lift1 (\i -> i+1)
```

Esta definición es mejor, pero todavía no es idónea. Primero, añadimos `lift2`:

```
lift2                   :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f                 =  \ra1 ra2 -> do a1 <- ra1
                                          a2 <- ra2
                                          step (f a1 a2)
```

Obsérvese que esta función explícitamente establece el orden de evaluación en la función elevada: el cómputo que devuelve `a1` ocurre antes que el cómputo para `a2`.

Usando `lift2`, podemos crear una nueva versión de == en la mónada `R`:

```
(==*)                   :: Ord a => R a -> R a -> R Bool
(==*)                   =  lift2 (==)
```

Hemos tenido que usar un nombre un poco diferente para esta función ya que == ya está definido, pero en algunos casos podemos usar el mismo nombre para las funciones elevadas y la original. Esta declaración de instancia permite que todos los operadores de la clase `Num` puedan ser usados en la mónada `R`:

```
instance Num a => Num (R a) where
  (+)                   =  lift2 (+)
  (-)                   =  lift2 (-)
  negate                =  lift1 negate
  (*)                   =  lift2 (*)
  abs                   =  lift1 abs
  fromInteger           =  return . fromInteger
```

La función `fromInteger` se aplica implícitamente a todas las constantes enteras en un programa Haskell (véase la sección 10.3); esta definición permite que las constantes enteras puedan actuar con el tipo `R Integer`. Podemos ahora, por fin, escribir la función de incremento de un modo completamente natural:

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

Obsérvese que no podemos elevar la clase `Eq` del mismo modo que la clase `Num`: el tipo de `==*` no es compatible con las posibles sobrecargas de `==` ya que el resultado de `==*` tiene tipo `R Bool` en lugar de `Bool`.

Para poder expresar cómputos interesantes en `R` necesitaremos una condicional. Como no podemos usar `if` (requiere que la condición tenga tipo `Bool` en lugar de `R Bool`), definimos la función `ifR`:

```
ifR                     :: R Bool -> R a -> R a -> R a
ifR tst thn els         =  do t <- tst
                              if t then thn else els
```

Ahora estamos preparados para escribir un programa más extenso en la mónada `R`:

```
fact                    :: R Integer -> R Integer
fact x                  =  ifR (x ==* 0) 1 (x * fact (x-1))
```

Aunque ésta no es la definición habitual de la función factorial es bastante leíble. La idea de proporcionar nuevas definiciones para operaciones existentes como `+` o `if` es una parte esencial para crear un lenguaje embebido en Haskell. Las mónadas son especialmente útiles para encapsular la semántica de estos lenguajes embebidos de un modo claro y modular.

Ahora estamos preparados para ejecutar algunos programas. Esta función ejecuta un programa en `R` dado un número máximo de pasos computacionales:

```
run                     :: Resource -> R a -> Maybe a
run s (R p)             =  case (p s) of
                             (_, Left v) -> Just v
                             _           -> Nothing
```

Hemos usado el tipo `Maybe` para tratar la posibilidad de que el cómputo no termine con el número de pasos reservados. Podemos ahora computar

```
run 10 (fact 2)  => Just 2

run 10 (fact 20) => Nothing
```

Podemos, por último, añadir alguna funcionalidad más interesante a la mónada. Consideremos la siguiente función:

```
(|||)                   :: R a -> R a -> R a
```

que ejecuta dos cómputos en paralelo, devolviendo el valor del primero que acabe. Una posible definición de esta función es:

```
c1 ||| c2                =  oneStep c1 (\c1' -> c2 ||| c1')
   where
        oneStep          :: R a -> (R a -> R a) -> R a
        oneStep (R c1) f =
```

```
R (\r -> case c1 1 of
            (r', Left v) -> (r+r'-1, Left v)
            (r', Right c1') ->  -- r' must be 0
             let R next = f c1' in
               next (r+r'-1))
```

Se lleva a cabo un paso de `c1`, devolviendo su valor si se completo el cómputo o, si `c1` devuelve un cómputo suspendido (`c1'`), se evalúa `c2 ||| c1'`. La función `oneStep` lleva a cabo un único paso de su argumento, devolviendo bien un valor evaluado o pasando el resto del cómputo a `f`. La definición de `oneStep` es simple: pasa a `c1` un 1 como recurso. Si se alcanza un valor final, éste es devuelto, ajustando el contador de pasos (es posible que un cómputo retorne sin llevar a cabo ningún paso por lo que el contador de recursos devuelto no ha de ser necesariamente 0). Si la computación queda suspendida, un contador de recursos enmendado es pasado a la continuación final.

Podemos ahora evaluar expresiones como `run 100 (fact (-1) ||| (fact 3))` sin que ésta no acabe ya que los dos cómputos son intercalados. (Obsérvese que nuestra definición de `fact` no acaba para `-1`). Podemos llevar a cabo muchas variaciones sobre esta estructura básica. Por ejemplo, podríamos extender el estado para incluir una traza de los pasos computados. También podríamos embeber esta mónada en la mónada estandarizada `IO`, de modo que los cómputos en `R` pudiesen interactuar con el mundo externo. Aunque este ejemplo es quizás más avanzado que otros en este tutorial, permite ilustrar la potencia de las mónadas como una herramienta para definir la semántica básica de un sistema. Este ejemplo también es un modelo de un pequeño *Lenguaje de Dominio Específico*, algo para los que Haskell es especialmente adecuado. Muchos otros LDEs han sido desarrollados en Haskell; véase `haskell.org` para más ejemplos. Especialmente interesantes son Fran, un lenguaje de animaciones reactivas, y Haskore, un lenguaje para la música por ordenador.

10 Números

Haskell proporciona una rica colección de tipos numéricos basados en los definidos en el lenguaje Scheme[7], los cuales a su vez están basados en los del lenguaje Common Lisp [8]. (Estos lenguajes, sin embargo, tienen tipado dinámico). Los tipos estándar incluyen enteros de precisión fija y arbitraria, ratios y números racionales formados desde cada tipo entero, así como reales y complejos de simple y doble precisión. Nos referiremos en este apartado a las características básicas de las estructuras de clases numéricas; para más detalles consúltese (§6.4).

10.1 Estructura de las Clases Numéricas

Las clases numéricas (la clase `Num` y aquellas que caen bajo ella) abarcan la mayoría de las clases estándar. También se observa que `Num` es una subclase de `Eq`, pero no de `Ord`; esto se debe a que la ordenación no tiene sentido para ciertos números, como los complejos. La subclase `Real` de `Num` es, sin embargo, una subclase de `Ord`.

La clase `Num` proporciona las operaciones básicas comunes a todos los tipos numéricos: éstas incluyen, entre otras, la adición, substracción, negación, multiplicación y el valor absoluto:

```
(+),(-),(*)  :: (Num a) => a -> a -> a
negate, abs  :: (Num a) => a -> a
```

[`negate` es una función que se utiliza como el operador prefijo *menos*; no podemos utilizar para ello `(-)`, porque esta es la función de substracción. Por ejemplo, `-x*y` es equivalente a `negate (x*y)`. (El menos prefijo tiene la misma precedencia que el menos infijo, la cual, por supuesto, es menor que la de la multiplicación.)]

Obsérvese que la clase `Num` no proporciona la operación de división; en dos subclases de `Num` se proporcionarán versiones diferentes de este operador de división.

La clase `Integral` proporciona una división entera y un resto de la división. Las instancias de `Integral` son `Integer` (una versión de enteros no limitados, también conocidas como "números grandes") e `Int` (limitados a la representación fija de la máquina para los enteros, con un rango equivalente al menos a 29 dígitos binarios más el signo). Una implementación particular de Haskell puede proveer otros tipos integrales además de estos. Obsérvese que `Integral` es una subclase de `Real`, en lugar de serlo de `Num` directamente; esto quiere decir que no hay intención de proporicionar enteros Gausianos.

Todos los demás tipos numéricos caen en la clase `Fractional`, la cual proporciona la operación de división ordinaria `(/)`. La subclase `Floating` proporciona funciones trigonométricas, logarítmicas y exponenciales.

La subclase `RealFrac` de `Fractional` y `Real` proporciona una función `properFraction`, la cual descompone un número en su parte entera y fraccionaria, y una colección de funciones que redondean a integral valores de diferentes formas:

```
properFraction          :: (Fractional a , Integral b) => a -> (a,b)
truncate, round,
```

```
floor, ceiling          :: (Fractional a , Integral b) => a -> b
```

La subclase `RealFloat` de `Floating` y `RealFrac` proporciona algunas funciones especializadas para el acceso a los componentes de un número flotante, el exponente y la mantisa. Los tipos `Float` y `Double` caen dentro de la clase `RealFloat`.

10.2 Constructores de Números

De los tipos numéricos estándar, `Int`, `Integer`, `Float` y `Double` son primitivos. Los otros se construyen a partir de estos.

`Complex` (se encuentra en la librería `Complex`) es un constructor de tipo que genera un dato complejo en la clase `Floating` a partir de datos de tipo `RealFloat`.

```
data (RealFloat a) => Complex a = !a :+ !a  deriving (Eq,Show)
```

El símbolo `!` es una anotación que hace al dato estricto; esto se discute en la Sección 6.3. Obsérvese que `(RealFloat a)` restringe el tipo del argumento; así, los complejos estándar tienen el tipo `Complex Float` y `Complex Double`. Desde la declaración `data` podemos ver que un número complejo se escribe como `x :+ y`; los argumentos definen la parte real e imaginaria respectivamente. Al ser`:+` un constructor de datos, podemos usarlo como patrón:

```
conjugate            :: (RealFloat a) => Complex a -> Complex a
conjugate (x :+ y)   =  x :+ (-y)
```

Similarmente, el tipo `Ratio` (se encuentra en la librería `Rational`) crea un tipo racional en la clase `RealFrac` desde instancias de tipo `Integral`. (`Rational` es un sinónimo de tipos para `Ratio Integer`). `Ratio`, sin embargo, es un constructor de tipos abstracto. En lugar de un constructor como `:+`, los racionales usan la función `%` para formar un ratio entre dos enteros. En lugar de utilizar patrones, los componentes pueden extraerse a través de funciones:

```
(%)                     :: (Integral a) => a -> a -> Ratio a
numerator,denominator   :: (Integral a) => Ratio a -> a
```

¿A qué se debe esta diferencia?. Los números complejos en forma cartesiana tiene una representación única. Por otro lado, los ratios no tiene representación única, pero tienen una forma canónica (forma reducida) que la implementación del tipo abstracto debe mantener; no es verdad que el valor de `numerator (x%y)` sea `x`, pero si es verdad que la parte real de `x+:y` es `x`.

10.3 Promociones Numéricas y Literales Sobrecargados

El Prelude y las librerías proporcionan varias funciones sobrecargadas que pueden utilizarse para la promoción explícita:

```
fromInteger         :: (Num a) => Integer -> a
fromRational        :: (Fractional a) => Rational -> a
toInteger           :: (Integral a) => a -> Integer
toRational          :: (RealFrac a) => a -> Rational
```

```
fromIntegral        :: (Integral a, Num b) => a -> b
fromRealFrac        :: (RealFrac a, Fractional b) => a -> b
fromIntegral        =  fromInteger . toInteger
fromRealFrac        =  fromRational . toRational
```

Dos de éstas se usan de forma implícita para la sobrecarga de literales: un número entero (sin punto decimal) es equivalente a la aplicación de `fromInteger` al valor del número como `Integer`. Similarmente, un numero flotante (con un punto decimal) es visto como una aplicación de `fromRational` al valor del número como `Rational`. Es decir, `7` tiene el tipo `(Num a) = > a`, y `7.3` tiene el tipo `(Fractional a) => a`. Esto quiere decir que podemos utilizar literales numéricos en funciones numéricas genéricas, por ejemplo:

```
halve               :: (Fractional a) => a -> a
halve x             = x * 0.5
```

Esta forma de sobrecargar los números tiene la ventaja adicional de que el método para interpretar un literal numérico como un número de un tipo dado puede ser especificado en la declaración de la instancia `Integral` o `Fractional` (ya que `fromInteger` y `fromFractional` son operadores de estas clases, respectivamente). Por ejemplo, la instancia `Num` de `(RealFloat a) => Complex a` contiene el método:

```
fromInteger x      = fromInteger x:+ 0
```

Esto nos dice que `fromInteger` está definida en `Complex` para producir un número complejo cuya parte real es proporcionada por la función `fromInteger` de la clase `RealFloat`. De esta manera, cada tipo definido por el usuario (por ejemplo, cuaterniones) puede hacer uso de los literales sobrecargados.

Como otro ejemplo, volvemos a ver la definición de `inc` de la Sección 2:

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

Ignorando la declaración de tipo, el tipo más general de `inc` es `(Num a) => a -> a`. La declaración de tipo explícita es legal pues es más específica que el tipo principal (uno más general causaría un error de tipos). La declaración de tipo tiene el efecto de restringir el tipo de `inc` y en este caso, algo como `inc (1::Float)` tendría un tipo erróneo.

10.4 Tipos Numéricos por Defecto

Considérese la siguiente definición de función:

```
rms             :: (Floating a) => a -> a -> a
rms x y         = sqrt ((x^2 + y^2)*0.5)
```

La función de exponenciación `(^)` (una de las tres formas diferentes de exponenciación en Haskell, véase §6.8.5) tiene el tipo `(Num a, Integral b) => a -> b -> a`, y, siendo `2` de tipo `(Num a) => a`, el tipo de `x^2` es `(Num a, Integral b) => a`. Esto es un problema; no hay forma de resolver la sobrecarga asociada con el tipo variable `b`, que aparece en el contexto aunque no se utiliza en la expresión de tipo. Esencialmente, el

programador ha especificado que la `x` debe elevarse al cuadrado pero no a cual de los dos tipos, `Int` o `Integer` pertenece el exponente. Por supuesto, podemos fijar esto:

```
rms x y        = sqrt ((x^(2::Integer) + y^(2::Integer))*0.5)
```

Sin embargo, es obvio que este tipo de soluciones son fastidiosas. Por otro lado, este tipo de ambigüedad en la sobrecarga no ocurre sólo en los números:

```
show (read "xyz")
```

¿Qué tipo tiene la cadena que se ha leído? Este problema es más serio que la ambigüedad en la exponenciación, porque allí, cualquier instancia de la clase `Integral` vale, mientras que aquí se pueden esperar comportamientos muy distintos dependiendo de la instancia de `Show` que se use para resolver la ambigüedad.

Debido a la diferencia entre los casos numéricos y generales en lo que se refiere a la ambigüedad de la sobrecarga, Haskell proporciona una solución restringida al caso numérico: cada módulo puede contener una declaración por defecto, consistente en la palabra clave `default` seguida de una serie de tipos monomórficos numéricos (tipos sin variables) separados por comas y encerrados entre paréntesis. Cuando una variable de tipo es ambigua, se consulta la lista por defecto, y se usa el primer tipo de la lista que satisface el contexto de esa variable de tipo. Por ejemplo, si la declaración por defecto es `default (Int,Float)`, el exponente ambiguo del ejemplo anterior será resuelto al tipo `Int`. (ver §4.3.4 para más detalles).

El valor por defecto de la clave `default` (si no se especifica nada) es `(Integer,Double)`, aunque `(Integer, Rational, Double)` podría también haber sido apropiado. Muchos programadores cautos preferirán usar `default ()` que no provee ningún valor por defecto.

11 Módulos

Un programa Haskell consta de una colección de *módulos*. Un módulo en Haskell responde al doble propósito de controlar el espacio de nombres y de crear tipos abstractos de datos.

El nivel superior de un módulo contiene cualquier tipo de declaraciones que ya hemos discutido: declaraciones de modo, declaraciones `data` y `type`, declaraciones de clase y de instancias, declaraciones de tipos, definiciones de función, y enlaces a través de patrones. A excepción del hecho de que las declaraciones de importación (descritas posteriormente) deben aparecer en primer lugar, el resto de declaraciones pueden aparecer en cualquier orden (el ámbito a nivel superior es mutuamente recursivo).

El diseño de módulos de Haskell es relativamente tradicional: el espacio de nombres de los módulos es totalmente plano, y los módulos no son "de primera categoría". Los nombres de los módulos son alfanuméricos y deben comenzar con una letra mayúscula. No hay conexión formal entre un módulo de Haskell y el sistema de ficheros que (típicamente) lo contiene. En particular, no hay conexión entre los nombres de los módulos y los nombres de los archivos, y más de un módulo podría residir en un solo fichero (un módulo puede incluso definirse entre varios ficheros). Por supuesto, una implementación adoptará muy probablemente convenciones que hagan la conexión entre los módulos y los ficheros más rigurosa.

Técnicamente hablando, un módulo es realmente una declaración que comienza con la palabra clave `module`; he aquí un ejemplo de un módulo con nombre `Tree`:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

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

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

El tipo `Tree` y la función `fringe` deben ser familiares; fueron dados como ejemplos en la sección 2.2.1. [debido a la palabra clave `where`, las reglas de sangrado están activas en el nivel superior de un módulo, y las declaraciones deben escribirse en la misma columna (típicamente la primera). Observe también que el nombre del módulo es igual que el del tipo; esto está permitido.]

Este módulo *exporta* explícitamente `Tree`, `Leaf`, `Branch`, y `fringe`. Si la lista de exportación del módulo se omite, se considera que se utiliza *all* y todos los nombres ligados en el nivel superior del módulo serán exportados. (En el ejemplo anterior todo se exporta explícitamente, así que el efecto sería el mismo.) Obsérvese que el nombre de un tipo y de sus constructores tienen que estar agrupados, como en `Tree(Leaf, Branch)`. Como simplificación, podríamos también escribir `Tree(..)`. También es posible la exportación de un subconjunto de los constructores. Los nombres en una lista de exportación no han de ser locales al módulo de exportación; cualquier nombre en el ámbito se puede enumerar en una lista de exportación

El módulo `Tree` puede ahora ser *importado* por cualquier otro módulo:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Los elementos que son importados/exportados a/desde un módulo se llaman *entidades*. Obsérvese la lista de importación explícita en el declaración de importación; omitirla causaría que todas las entidades exportadas por `Tree` fuesen importadas.

11.1 Nombres cualificados

Existe un problema obvio con los nombres importados en el espacio de nombres de un módulo. ¿Qué ocurre si dos módulos importados contienen diversas entidades con el mismo nombre? Haskell soluciona este problema usando *nombres cualificados*. Una declaración de importación puede utilizar la palabra clave `qualified` para hacer que los nombres importados sean precedidos por el nombre del módulo de donde se importaron. Estos prefijos son seguidos por el carácter`.' sin espacios. [ los nombres cualificados son parte de la sintaxis léxica. Así, `A.x` y `A . x` son absolutamente diferentes: el primero es un nombre cualificado y el segundo un uso de la función '.' ] Por ejemplo, usando el módulo `Tree` anterior:

```
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]   -- Una   definicion   diferente   de   fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

module Main where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Algunos programadores de Haskell prefieren utilizar todas las entidades importadas de un modo cualificado, haciendo el origen de cada nombre explícito en cada uso. Otros prefieren nombres cortos y utilizan solamente nombres cualificados cuando son absolutamente necesarios.

Los cualificadores suelen usarse para resolver conflictos entre diversas entidades que tengan el mismo nombre. Pero ¿qué ocurre si la misma entidad se importa a través de varios módulos? Afortunadamente, se permiten tales choques: una entidad se puede importar por varias rutas sin conflicto. El compilador sabe si las entidades de diversos módulos son realmente iguales.

11.2 Tipos Abstractos de Datos

Aparte del control del espacio de nombres, los módulos proporcionan la única manera de construir tipos abstractos de datos (TAD) en Haskell. Por ejemplo, la característica de un TAD es que *la representación del tipo* está *oculto*; todas las operaciones sobre el TAD se hacen en un nivel abstracto que no depende de la representación. Por ejemplo, aunque el

tipo `Tree` es bastante simple como para que lo hagamos abstracto, un TAD para él puede incluir las operaciones siguientes:

```
data Tree a             -- el          nombre          del          tipo
leaf                    :: a -> Tree a
branch                  :: Tree a -> Tree a -> Tree a
cell                    :: Tree a -> a
left, right             :: Tree a -> Tree a
isLeaf                  :: Tree a -> Bool
```

Un módulo que implementa esto es:

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

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

leaf                    = Leaf
branch                  = Branch
cell  (Leaf a)          = a
left  (Branch l r)      = l
right (Branch l r)      = r
isLeaf   (Leaf _)       = True
isLeaf   _              = False
```

Obsérvese que en la lista de exportación, el nombre del tipo `Tree` aparece sólo (sin sus constructores). Así `Leaf` y `Branch` no son exportados, y la única manera de crear y manejar árboles fuera de este módulo es a través de las operaciones abstractas exportadas. Una ventaja de esta ocultación es que posteriormente podemos *cambiar* la representación del tipo sin afectar a los usuarios del mismo.

11.3 Más Características

He aquí una breve revisión de otros aspectos del sistema de módulos. Para más detalles, consultase el Informe.

- Una declaración `import` puede ocultar selectivamente entidades utilizando la cláusula `hiding` en la declaración de importación. Esto puede ser usado para excluir nombres que son importados de otro módulo y evitar así la necesidad de cualificadores.
- Una declaración `import` puede contener una cláusula `as` para especificar un cualificador distinto al nombre correspondiente del modulo importado. Esto es útil para acortar los cualificadores de módulos con nombres largos o para adaptar fácilmente un cambio en el nombre de un módulo sin cambiar todos los cualificadores.
- Todos los programas importan implícitamente el módulo `Prelude`. Una importación explícita de Prelude anula la implícita por completo. Así,

  ```
  import Prelude hiding length
  ```

  no importará `length` desde el Standard Prelude, permitiendo que el nombre `length` sea definido de forma diferente.
- Las declaraciones de instancias no pueden ser incluidas en la lista de exportaciones o importaciones de un módulo. Cada módulo exporta todas sus declaraciones de instancias y cada importación hace visible todas las declaraciones de instancias.

- La exportación de los métodos de clase puede hacerse usando cualquiera de las dos formas definidas para los constructores de clase, entre paréntesis trás el nombre de la clase o como variables ordinarias.

Aunque el sistema de módulos de Haskell es relativamente tradicional, existen muchas reglas relativas a la importación y exportación de valores. La mayoría de ellas son obvias--- por ejemplo, es ilegal importar dos entidades diferentes con el mismo nombre en el mismo ámbito. Otras reglas no son tan obvias---por ejemplo, para un tipo y clase dados, no puede haber más de una combinación de la clase y del tipo en cualquier lugar del programa. El lector debería leer el Informe para más detalles (§5).

12 Errores de tipos más comunes

Esta sección muestra una descripción intuitiva de algunos problemas comunes que los principiantes suelen tener cuando se enfrentan al sistema de tipos de Haskell.

12.1 Polimorfismo restringido en construcciones let

Cualquier lenguaje que usa el sistema de tipos de Hindley-Milner tiene el llamado *polimorfismo restringido en construcciones let*, porque los identificadores no ligados por las cláusulas `let` o `where` (o en el nivel superior de un módulo) están limitados con respecto a su polimorfismo. En particular, una función *lambda-bound* (es decir, pasada como argumento a otra función) no puede ser instanciada de dos diferentes maneras. Por ejemplo, este programa es ilegal:

```
let f g  =  (g [], g 'a')                        -- expresion  mal  tipada
in f (\x->x)
```

porque `g`, está ligada por una lambda abstracción cuyo tipo principal es `a->a`, y se usa dentro de `f` de dos formas diferentes con tipos `[a]->[a]` la una y `Char->Char` la otra.

12.2 Sobrecarga Numérica

Es fácil olvidarse de que los números *están sobrecargados*, y *no hay promoción implícita* a los distintos tipos numéricos, como en muchos otros lenguajes. Hay expresiones numéricas genéricas que a veces no son tan genéricas. Un error numérico común es el siguiente:

```
average xs              =  sum xs / length xs           -- Mal!
```

`(/)` requiere argumentos fraccionarios, pero el resultado de `length` es `Integer`. La discordancia del tipo se debe corregir con una promoción explícita:

```
average                 :: (Fractional a) => [a] -> a
average xs              =  sum xs / fromIntegral (length xs)
```

12.3 La Restricción Monomórfica

El sistema de tipos de Haskell contiene una restricción relacionada con las clases de tipos que no se encuentra en sistemas ordinarios de tipos basados en el de Hindley-Milner: *la restricción del monomorfismo*. La razón de esta restricción está relacionada con una ambigüedad sutil del tipo y se explica en profundidad en el informe (§4.5.5). Una explicación más simple es la siguiente:

La restricción del monomorfismo dice que cualquier identificador enlazado a través de un patrón (ligaduras a un solo identificador), y que no tiene ninguna declaración explícita de tipo, debe ser *monomórfica*. Un identificador es monomórfico si o bien no está sobrecargado, o lo está pero se utiliza a lo sumo de una forma sobrecargada y no se exporta.

Las violaciones de esta restricción dan lugar a un error de tipo estático. La manera más simple de evitar el problema es proporcionar una declaración explícita del tipo. Observe que *cualquier* declaración de tipo valdrá (siempre que el tipo sea correcto).

Una violación común de la restricción sucede con las definiciones de funciones de orden superior, como en esta definición de `sum` del Standard Preluder:

```
sum                     =  foldl (+) 0
```

Como se ve, esto causaría un error de tipo estático. Podemos resolver el problema agregando la declaración de tipo:

```
sum                     :: (Num a) => [a] -> a
```

Obsérvese que no se habría presentado este problema si hubiesemos escrito:

```
sum xs                  =  foldl (+) 0 xs
```

porque la restricción se aplica solamente a los patrones.

13 Tablas

Idealmente, las tablas en un lenguaje funcional deberían ser vistas simplemente como funciones de índices a valores, pero pragmáticamente, para asegurar el acceso eficiente a los elementos de la tabla, necesitamos estar seguros de que podemos aprovecharnos de las características especiales de los dominios de estas funciones, que son isomorfas a los subconjuntos contiguos finitos de los números enteros. Haskell, por lo tanto, no trata las tablas como funciones generales con una operación de aplicación, sino como tipos abstractos de datos con una operación de indexación.

Podemos distinguir dos aproximaciones principales a las tablas funcionales: definición *incremental y monolítica*. En el caso incremental, tenemos una función que produce una tabla vacía de un tamaño dado y otra que tome una tabla, un índice, y un valor, produciendo una nueva tabla que se diferencie de la vieja solamente en el valor del índice. Obviamente, una implementación ingenua de tal semántica de la tabla sería intolerablemente ineficaz, requiriendo una nueva copia de la tabla para cada redefinición incremental; así, las implementaciones serias que usan esta aproximación emplean análisis estático sofisticado y mecanismos en tiempo de ejecución para evitar la copia excesiva. Por otro lado, la aproximación monolítica, construye una tabla de una vez, sin construcciones intermedias de la tabla. Aunque Haskell tiene un operador incremental para la actualización de la tabla, el uso principal de la tabla es monolítica.

Las tablas no son parte del Standard Prelude-- una librería estándar contiene las operaciones para tablas. Cualquier módulo que use tablas debe importar el módulo `Array`.

13.1 Tipos Índice

La librería `Ix` define una clase para los índices de las tablas:

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

Los declaraciones de instancias se proporcionan para `Int, Integer`, `Char`, `Bool`, y tuplas de tipos `Ix`; además, se pueden derivar instancias automáticamente para los tipos enumerados y tuplas. Debemos ver los tipos primitivos como índices vectoriales y las tuplas como índices de matrices rectangulares multidimensionales. Observe que el primer argumento de cada uno de las operaciones de la clase `Ix` es un par de índices; éstos son típicamente *los límites* (primero y último) de una tabla. Por ejemplo, los límites de un vector de 10 elementos, con origen el cero de `Int` serían `(0,9)`, mientras que una matriz de 100 por 100 con origen-1 debe tener los límites `((1,1),(100,100))`. (En muchos otros lenguajes, tales límites serían escritos en una forma como `1:100, 1:100`, pero la forma actual es mejor para el sistema de tipos, puesto que cada límite es del mismo tipo que el índice general.)

La operación `range` toma un par de límites y produce la lista de los índices que caen entre esos límites, en el orden del índice. Por ejemplo,

```
range (0,4) => [0,1,2,3,4]
```

```
range ((0,0),(1,2)) => [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]
```

El predicado `inRange` determina si un índice cae entre un par de límites dados. (Para un tipo tupla, este predicado es evaluado componente a componente.) Finalmente, la operación `index` es necesaria para acceder a un elemento determinado del array: dado el par de límites y un índice dentro del rango, la operación devuelve el ordinal del índice partiendo del origen dentro del rango; por ejemplo:

```
index (1,9) 2 => 1

index ((0,0),(1,2)) (1,1) => 4
```

13.2 Creación de una Tabla

La función de creación de una tabla monolítica Haskell construye una tabla a partir de un par de límites y una lista de pares conteniendo índices y valores (una *lista de asociaciones*):

```
array                   :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

He aquí, por ejemplo, la definición de una tabla con los cuadrados de los números del 1 al 100

```
squares                 =  array (1,100) [(i, i*i) | i <- [1..100]]
```

El uso de listas por comprensión para la lista de asociaciones es típico; además, estas expresiones se parecen a las tablas por *comprensión* utilizados en el lenguaje Id [6].

El acceso a un elemento de la tabla se realiza a través del operador `!`, y los límites pueden ser extraidos con la función `bounds`:

```
squares!7 => 49

bounds squares => (1,100)
```

Podemos generalizar este ejemplo, parametrizando los límites y la función a aplicar a cada índice en la creación:

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

Así, podemos definir `squares` como `mkArray (\i -> i * i) (1,100)`.

Muchas tablas se definen recursivamente, esto es, con los valores de unos elementos dependiendo de los valores de otros. Aquí, por ejemplo, tenemos una función que devuelve una tabla con los números de Fibonacci:

```
fibs    :: Int -> Array Int Int
fibs n  =  a  where a = array (0,n) ([(0, 1), (1, 1)] ++
                                     [(i, a!(i-2) + a!(i-1)) | i <-
 [2..n]])
```

Otro ejemplo de tal recurrencia es la matriz del "frente de onda" *n* por *n* en la cual, los elementos de la primera fila y la primera columna tienen todos el valor 1 y cualquier otro elemento es la suma de sus vecinos del oeste, noroeste y norte:

```
wavefront       :: Int -> Array (Int,Int) Int
wavefront n     =  a  where
                   a = array ((1,1),(n,n))
                        ([((1,j), 1) | j <- [1..n]] ++
                         [((i,1), 1) | i <- [2..n]] ++
                         [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                                     | i <- [2..n], j <- [2..n]])
```

La matriz del frente de onda es llamada así porque en una implementación paralela, la repetición dicta que el cómputo puede comenzar con la primera fila y columna en paralelo y proceder en forma de cuña, viajando del noroeste al sureste. Es importante observar, sin embargo, que no se especifica ningún orden del cómputo en la lista de asociaciones.

En cada uno de nuestros ejemplos, hemos dado una única asociación a cada índice de la tabla y sólo para los índices que caen dentro de los límites del mismo, y ciertamente, debemos hacer esto en general para que una tabla quede perfectamente definida. Una lista de asociaciones con índices fuera de rango producen un error; si un índice no aparece o aparece más de una vez, no se produce un error de inmediato pero el valor de la tabla en ese índice queda indefinido, de tal manera que un acceso a ese índice en la tabla producirá un error.

13.3 Acumulación

Podemos relajar la restricción de que un índice aparezca más de una vez en la lista de asociaciones especificando cómo combinar múltiples valores asociados a un mismo índice. El resultado es una tabla *acumulativa*:

```
accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -
> Array a b
```

El primer argumento de un `accumArray` es la *función de acumulación*, el segundo es el valor inicial (el mismo para todos los elementos de la tabla), y el resto de argumentos son los límites y la lista de asociaciones, como en la función `array`. Típicamente, la función de acumulación es `(+)`, y el valor inicial es, cero; por ejemplo, esta función toma un par de límites y una lista de valores (de tipo índice) y devuelve un histograma; esto es, una tabla con el número de ocurrencias de cada valor dentro de los límites:

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <-
 is, inRange bnds i]
```

Supongamos que tenemos una colección de medidas en un intervalo *[a,b)*, y queremos dividir el intervalo en diez partes y contar el número de medidas en cada una:

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade
                   where decade x = floor ((x - a) * s)
                         s        = 10 / (b - a)
```

13.4 Actualización incremental

Además de una función de creación monolítica de tablas, Haskell tiene una función de actualización incremental, escrita como un operador infijo `//`; el caso más simple corresponde a una tabla a cuyo elemento de la posición `i` se actualiza a `v`, se escribe como `a // [(i, v)]`. Los delimitadores usados se deben a que el argumento derecho es una lista de asociaciones conteniendo un subconjunto propio de índices de la tabla:

```
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

Como con la función `array`, los índices en la lista de asociaciones deben ser únicos para los valores que estén definidos. Por ejemplo, la función que intercambia dos filas de una matriz sería:

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -
> Array (a,b) c
swapRows i i' a =  a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                        [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

La concatenación aquí de dos listas por comprensión sobre la misma lista de índice `j` es, sin embargo, levemente ineficiente; es igual que escribir dos bucles en lugar de uno en un lenguaje imperativo. No tema, ya que podemos realizar el equivalente a una optimización por la fusión de bucles en Haskell:

```
swapRows i i' a =  a // [assoc | j <- [jLo..jHi],
                                 assoc <- [((i ,j), a!(i',j)),
                                           ((i',j), a!(i, j))] ]
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

13.5 Un ejemplo: Multiplicación de matrices

Completamos nuestra introducción a las tablas en Haskell con el ejemplo familiar de la multiplicación de matrices, utilizando ventajosamente la sobrecarga para construir una función suficientemente general. Estando involucradas la suma y multiplicación de los elementos de las matrices, tomaremos una función que multiplique cualquier tipo numérico, a no ser que nos empeñemos en lo contrario. Adicionalmente, si tenemos cuidado de aplicar sólo `(!)` y las operaciones de `Ix` a los índices, tendremos genericidad sobre el tipo de los índices, y en efecto, los tipos índices de las cuatro filas y columnas no necesitan ser iguales. Sin embargo, por simplicidad, supondremos que los índices de la columna de la izquierda y la fila de la derecha tienen el mismo tipo, y además, sus límites son iguales:

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                   Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =  array resultBounds
                     [((i,j), sum [x!(i,k) * y!(k,j) | k <-
 range (lj,uj)])
                                          | i <- range (li,ui),
                                            j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))          =  bounds x
              ((li',lj'),(ui',uj'))      =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')     =  ((li,lj'),(ui,uj'))
```

```
                | otherwise             = error "matMult: límites
incompatibles"
```

De otra forma, podemos definir `matMult` usando `accumArray`, resultando una representación que recuerda a la formulación usual en un lenguaje imperativo:

```
matMult x y     =  accumArray (+) 0 resultBounds
                              [((i,j), x!(i,k) * y!(k,j))
                                      | i <- range (li,ui),
                                        j <- range (lj',uj')
                                        k <- range (lj,uj)  ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: límites
incompatibles"
```

Podemos generalizar aún más haciendo la función de orden superior, simplemente remplazando `sum` y `(*)` por parámetros funcionales:

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult f g x y  =  array resultBounds
                       [((i,j), f [x!(i,k) `g` y!(k,j) | k <-
 range (lj,uj)])
                                  | i <- range (li,ui),
                                    j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: límites
incompatibles"
```

Los aficionados a APL reconocerán la utilidad de funciones como las siguientes:

```
genMatMult maximum (-)
genMatMult and (==)
```

En la primera de ellas, los argumentos son matrices numéricas, y el elemento *(i,j)* del resultado es la máxima diferencia entre los correspondientes elementos de la fila *i*-ésima y la columna *j*-ésima de las entradas. En la segunda, los argumentos son matrices de cualquier tipo con igualdad, y el resultado es una matriz booleana en la que el elemento *(i,j)* es `True` sí y sólo sí la *i*-ésima fila del primer argumento y la *j*-ésima columna del segundo son iguales como vectores.

Obsérvese que los tipos de los elementos de `genMatMult` no necesitan ser iguales, sino apropiados para la función parámetro `g`. Podemos generalizar aún más eliminando el requerimiento de que los tipos índices de la primera columna y de la segunda fila sean los mismos, claramente, dos matrices son consideradas "multiplicables" si el número de columas de la primera y el número de filas de la segunda son iguales. El lector puede intentar derivar esta versión más general. (**Indicación:** Use la operación `index` para determinar las longitudes.)

14 La Siguiente Etapa

Una gran colección de recursos de Haskell están disponibles en la Web en `haskell.org`. Aquí encontrará compiladores, demos, documentos, y mucha información valiosa sobre Haskell y la programación funcional. Los compiladores o los intérpretes de Haskell se ejecutan en casi todos los equipos y sistemas operativos. El sistema Hugs es pequeño y portable -- es un vehículo excelente para aprender Haskell.

15 Bibliografía

[1] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, New York, 1998.

[2] A.Davie. *Introduction to Functional Programming System Using Haskell*. Cambridge University Press, 1992.

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

[4] Simon Peyton Jones (editor). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106*, Feb 1999.

[5] Simon Peyton Jones (editor) The Haskell 98 Library Report. *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1105*, Feb 1999.

[6] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[7] J. Rees and W. Clinger (eds.). The revised$_3$ report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37--79, December 1986.

[8] G.L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Mass., 1984.

[9] P. Wadler. How to replace failure by a list of successes. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201*, pages 113--128. Springer Verlag, 1985.

[10] P. Wadler. Monads for Functional Programming In *Advanced Functional Programming* , Springer Verlag, LNCS 925, 1995.