

La programación funcional en Haskell

Ricardo Peña Marí
Departamento de Informática y Automática
Universidad Complutense de Madrid
e-mail: ricardo@dia.ucm.es

Report DIA-95/2, 7 de Febrero de 1995

1 Introducción

Todavía demasiados informáticos siguen identificando la programación funcional con la programación en Lisp, el primer lenguaje funcional, creado por John McCarthy en 1959 y casi el primer lenguaje de alto nivel, coetáneo de Fortran. Desde entonces hasta aquí han sucedido muchas cosas en el mundo de la programación, y en particular en el de la programación funcional. Este trabajo describe las características más relevantes de los lenguajes funcionales actuales y tratará de mostrar que, para muchos usos, pueden sustituir con ventaja a los lenguajes imperativos convencionales. Al mismo tiempo, pretende servir de vehículo de difusión del lenguaje de programación Haskell [HPJW92]. Este lenguaje representa el estado del arte en el campo de la programación funcional y ha sido diseñado con el objetivo de introducir el paradigma funcional en el mayor número de ámbitos posibles, incluyendo por supuesto el de la práctica industrial de la programación.

Los paradigmas de programación funcional y lógica han surgido de un modo diferente al paradigma imperativo. En el último caso, los llamados lenguajes de alto nivel (p.e. Ada) pueden considerarse como un recubrimiento de los aspectos más cercanos al hardware del computador. Este recubrimiento permite a los programadores elevar el nivel de abstracción de sus programas y poder pensar en términos de tipos de datos cercanos a su problema o escribir expresiones aritméticas, y de cualquier otro tipo, en una notación independiente de la máquina. Sin embargo, el modelo básico de cómputo continúa siendo el que se ha dado en llamar, en honor al creador de la arquitectura de los computadores actuales, *modelo Von Neumann*. En dicho modelo, un programa ha de detallar la *secuencia exacta* de instrucciones que el computador ha de seguir para modificar el estado de la memoria, desde el estado de partida hasta el estado final deseado. En dicha secuencia el programa pasa por sucesivos estados intermedios que el programador ha de tener en cuenta. Las instrucciones de un programa típico pueden así clasificarse en dos grandes grupos:

- Instrucciones que *modifican el estado* del cómputo. La más importante de ellas es la instrucción de asignación $x := Exp$, donde la variable x que cambia de estado puede estar o no subindicada.

- Instrucciones que *gobiernan el flujo* de control. Es decir, instrucciones iterativas y condicionales que alteran los subíndices de las instrucciones de asignación, deciden el número de repeticiones a realizar y si ha de pasarse o no a un nuevo cómputo.

La crítica a este modelo de cálculo fué realizada con mucha contundencia y acierto por John Backus en [Bac78]. Sólo repetiremos aquí su opinión acerca de la instrucción de asignación de los lenguajes imperativos por considerarla de una gran lucidez: según Backus, dicha instrucción merece el apelativo de *cuello de botella intelectual* que constriñe al programador a pensar en términos de una palabra al tiempo. Los grandes cambios en la memoria del computador que conlleva un cálculo complejo han de ser realizados en base a una instrucción capaz de producir tan sólo un pequeño cambio. Gran parte de la tarea del programador consiste así en organizar los subíndices de las variables objeto de las asignaciones y la secuencia en que han de realizarse tales pequeños cambios para producir el cambio global deseado.

Como veremos, en el modelo funcional el programador puede pensar en términos de unidades de información más grandes, y no ha de sobre-especificar la secuencia en que tienen lugar las acciones elementales dentro de la máquina. La libertad que el programador deja a este respecto puede ser utilizada por el compilador para mejorar la eficiencia de la ejecución. El trabajo de Backus [Bac78] marca el punto de inflexión hacia un renovado interés por los lenguajes funcionales, que condujo durante la década de los 80 a numerosas propuestas innovadoras. Los lenguajes funcionales modernos, y Haskell como culminación de todos ellos, son el fruto de este esfuerzo.

El paradigma funcional surge, como se ha dicho, de un modo diferente al imperativo. Aquí el modelo de cómputo procede de las matemáticas y es incluso históricamente anterior a la aparición de los computadores. Dicho modelo es el cálculo lambda, desarrollado por A. Church entre 1932 y 1941. El interés de Church era la definición precisa de las *funciones computables* que, en esencia, son las que se pueden programar en un computador. La “ejecución” de una expresión lambda consiste en su reducción a *forma normal* mediante un número finito (si el programa termina) de pasos de *reescritura*. Este proceso de reducción o de reescritura, muy similar a la idea familiar de simplificación de expresiones, puede ser llevado a cabo por un computador convencional mediante un intérprete apropiado, dando lugar así a un lenguaje de programación. La ejecución de un programa funcional consiste, en esencia, en la aplicación de una función a unos datos. La expresión que resulta se reduce mediante reescritura a sucesivas expresiones equivalentes, cada vez más sencillas, hasta llegar a una expresión que no es posible reducir más. Esta expresión irreducible se considera el resultado de la ejecución del programa para los datos de partida. Conviene añadir que la arquitectura de los computadores convencionales no resulta especialmente adaptada a este tipo de cómputo, por lo que cabe esperar una pérdida de eficiencia con respecto a la realización de la misma función en un lenguaje imperativo. Esta cuestión se analiza en detalle en la sección 3.

2 Características de la programación funcional moderna

La programación funcional incorpora características tomadas de diferentes fuentes junto con otras características originales que, en conjunto, dan al paradigma una gran potencia expresiva:

1. De la programación imperativa toma muchas de las facilidades presentes en los lenguajes modernos: disciplina fuerte de tipos, tipos predefinidos, tipos definibles por el programador, tipos abstractos de datos, módulos, compilación separada y otras.
2. De las matemáticas, y en particular de la lógica, toma su aspecto declarativo —las funciones se definen mediante ecuaciones— y propiedades algebraicas útiles para razonar, la más importante de las cuales es la de *transparencia referencial*, según la cual cada identificador o expresión denota un único valor independientemente del contexto en el que sean utilizados.
3. De los lenguajes de especificación algebraica toma la definición mediante patrones y algunas ideas sobre genericidad.

Además, incorpora conceptos originales como el de *polimorfismo* —entendido como polimorfismo paramétrico, no como el polimorfismo *ad hoc* de los lenguajes orientados a objetos—, el de funciones de *orden superior* y el mecanismo de *evaluación perezosa* que posibilita la definición de funciones y tipos de datos no estrictos y estructuras infinitas de datos. Muchas de las personas que han trabajado en programación funcional están de acuerdo en que estas dos últimas características —orden superior y

evaluación perezosa— son las más específicas del paradigma y las que confieren al mismo la mayor parte de su potencia y versatilidad (ver p.e. [Hug90]). Iremos detallando todas estas cuestiones al tiempo que introducimos gradualmente el modo funcional de programar.

2.1 Definición ecuacional

Un programa funcional consiste en esencia en un conjunto de definiciones de función. Cada definición de función consta de un conjunto de ecuaciones y en las mismas puede hacerse referencia a funciones definidas en otra parte del programa. En el caso más sencillo, una sola ecuación basta para definir una función. Por ejemplo,

```
triplo x = 3 * x
```

define una función `triplo` que devuelve el triple del valor de su único parámetro. La expresión $f e_1 \dots e_n$ denota la aplicación de la función f a las expresiones e_1, \dots, e_n (se supone que f ha sido definida con n parámetros formales). Así, `triplo (5+2)` denota la aplicación de la función `triplo` a la expresión `5+2`, dando como resultado el valor esperado 21. Este valor es la forma normal o irreducible de la expresión `triplo (5+2)`.

En otros casos pueden ser necesarias varias ecuaciones. Cada ecuación define el resultado de la función para un subconjunto de los valores de entrada. Dicho subconjunto se determina expresando los parámetros formales como *patrones*. A los parámetros reales que encajen con dichos patrones les será aplicable la ecuación. Por ejemplo, dos patrones muy comunes para el tipo de datos lista son `[]` y `x:xs`. El primero designa la lista vacía y el segundo las listas no vacías con elemento de cabeza `x` y lista resto `xs`. Al producirse el encaje con el parámetro real, las variables del patrón quedan vinculadas a valores. Así, la lista `[3,4,5]` encaja con el patrón `x:xs`, estableciéndose los vínculos `x=3` y `xs=[4,5]`. Las siguientes ecuaciones definen una función que suma los elementos de una lista de enteros:

```
suma [] = 0
suma (x:xs) = x + (suma xs)
```

La recursividad es el equivalente en programación funcional a la iteración en programación imperativa. Al igual que en esta última, una parte importante del razonamiento de corrección es asegurar que el programa termina.

La definición mediante patrones establece un análisis por casos sobre los parámetros de entrada. Dicho análisis puede hacerse aún más fino distinguiendo subcasos dentro de una ecuación por medio de *guardas*. Una guarda es una expresión booleana que, caso de ser cierta, hace que sea aplicable la parte derecha a ella asociada. Puede hacer referencia a las variables formales introducidas por los patrones de la parte izquierda de la ecuación. Por ejemplo, la siguiente definición,

```
insert x [] = [x]
insert x (y:ys)
  | x < y = x:y:ys
  | x == y = y:ys
  | otherwise = y:insert x ys
```

inserta un valor `x` en una lista ordenada crecientemente y sin repeticiones. La cláusula `otherwise` es una guarda que siempre se cumple (en el preludio de Haskell está definida mediante la ecuación `otherwise = True`) y se utiliza para recoger el caso en que no se cumplen las guardas anteriores. La función `==` compara dos enteros por igualdad. La aplicación de funciones tiene más prioridad que cualquier otro operador del lenguaje, por lo que la expresión `y:insert x ys` ha de leerse como `y:(insert x ys)`.

Por legibilidad, es conveniente que los patrones de las diversas ecuaciones que definen una función sean excluyentes entre sí e igualmente las guardas dentro de cada ecuación. No obstante, la semántica del lenguaje establece que los patrones y las guardas se ensayan secuencialmente según el orden en que han sido escritas. Una ecuación falla para unos parámetros reales si, o bien éstos no encajan en los patrones correspondientes, o bien encajan pero son falsas todas las guardas de la misma. Si una ecuación falla, se ensaya la siguiente. Si todas fallan, se produce un error en tiempo de ejecución. Considérese la siguiente definición de los números de Fibonacci:

```
fib 0 = 1
fib n | n == 1 = 1
      | n >= 2 = fib (n-1) + fib (n-2)
```

Si evaluamos la expresión `fib (-3)`, el valor `-3` no encaja con el patrón `0` de la primera ecuación por lo cual ésta falla. En cambio, sí encaja con el de la segunda ya que una variable es un patrón universal que encaja con cualquier valor. La ecuación falla a su vez al ser falsas todas las guardas. El resultado es **error**. Si sustituimos la segunda guarda por la cláusula `otherwise`, la evaluación de `fib (-3)` daría lugar a no terminación.

Algunas funciones se utilizan habitualmente en matemáticas como operadores infijos. En Haskell se pueden definir operadores infijos siempre que su nombre conste de uno o más símbolos no alfanuméricos. Así, la definición del `and` booleano está escrita en el preludio `standard` de Haskell del modo siguiente:

```
True  && x = x
False && _ = False
```

El símbolo `'_'` designa una variable anónima. Las variables anónimas se utilizan para aumentar la legibilidad. Indican al lector que su valor no interviene en la parte derecha de la ecuación.

Quizás es apropiado decir en este punto que una política general de los lenguajes funcionales es eliminar los tratamientos excepcionales: si algo está permitido para la parte predefinida del lenguaje, también lo está para la parte definida por el programador. Éste puede definir operadores infijos, establecer su prioridad de evaluación en expresiones ambiguas, decir si asocian hacia la izquierda, hacia la derecha, o que no son asociativos. La mayor parte de los tipos y facilidades predefinidas de Haskell están de hecho definidas en el llamado preludio `standard` que está escrito en Haskell y equivale a la librería del lenguaje. Si se cambiara dicho preludio, se tendrían otras facilidades predefinidas distintas.

Cualquier operador infijo puede utilizarse como función prefija de dos parámetros escribiéndolo entre paréntesis. A la inversa, toda función prefija de dos parámetros puede utilizarse como operador infijo escribiéndola entre comillas abiertas simples. Así:

```
(&&) e1 e2   equivale a   e1 && e2
x 'insert' xs equivale a   insert x xs
```

Frecuentemente, la expresión de la parte derecha de una ecuación resulta complicada o contiene subexpresiones repetidas. En estos casos, se puede incrementar la legibilidad y la modularidad de una definición por medio de las llamadas *definiciones locales*. Considérese la siguiente función que calcula las raíces de una ecuación de segundo grado de la forma $ax^2 + bx + c$:

```
raices a b c | dis == 0.0 = let r = (negate b) / (2.0*a) in (r,r)
              | dis > 0.0 = let
                              r1 = ((negate b) + rdis) / (2.0*a)
                              r2 = ((negate b) - rdis) / (2.0*a)
                          in
                              (r1,r2)
              | dis < 0.0 = error "raices imaginarias"
              where rdis = sqrt dis
                    dis  = b*b-4.0*a*c
```

El resultado es una tupla (r_1, r_2) de dos números reales, que coinciden cuando la ecuación tiene una sola raíz. La cláusula `where` introduce las definiciones de los identificadores `dis` y `rdis` que son locales a la parte derecha de la ecuación y a las guardas, mientras que las cláusulas `let` introducen definiciones, como las de `r`, `r1` y `r2`, locales a sus respectivas expresiones. Los identificadores definidos en ambos casos no son visibles fuera de las expresiones afectadas. Ambas cláusulas se pueden anidar. Su sintaxis precisa es:

```
expresión where {decl1;...;decln}
let {decl1;...;decln} in expresión
```

Las llaves `{, }` y los delimitadores `' ; '` son parte de la sintaxis concreta. Equivalen a los paréntesis **begin**, **end** y al `' ; '` de algunos lenguajes. Sin embargo, si se siguen ciertas normas en la disposición espacial del texto, pueden omitirse. Es decir, el sangrado del texto tiene significado sintáctico. Curiosamente, los lenguajes funcionales recuperan en este sentido algo del valor semántico que los primeros lenguajes de alto nivel (como p.e. Fortran) daban al formato. No obstante, si el programador deseara formato libre, siempre podría tenerlo empleando delimitadores explícitos. Obsérvese también en el ejemplo que un identificador puede utilizarse aunque su definición aparezca más adelante en el texto. En Haskell todas

las declaraciones se consideran simultáneas y su orden es irrelevante (con la excepción mencionada de la evaluación de patrones y guardas no excluyentes). Incluso pueden ser mutuamente recursivas como en el siguiente ejemplo que define las funciones `par` e `impar` para enteros no negativos:

```
par 0          = True
par x | x > 0 = impar (x-1)
impar 0       = False
impar x | x > 0 = par (x-1)
```

2.2 Funciones de orden superior

La divisa con la que se introducían los trabajos sobre programación funcional durante la década de los 80 era que “las funciones son ciudadanos de primera clase”. Ello significa que una función es un objeto más del lenguaje, tratado a todos los efectos como los otros objetos. En particular, las funciones pueden ser pasadas como argumento a otras funciones, pueden ser devueltas como resultado y pueden figurar en estructuras de datos. Las funciones que toman como argumento o/y devuelven como resultado una función se denominan de *orden superior*.

Las funciones en Haskell tienen un único argumento y devuelven un único resultado. Si f toma un argumento de tipo T_1 y devuelve un resultado de tipo T_2 , f es de tipo $T_1 \rightarrow T_2$, es decir, pertenece al conjunto de todas las funciones cuyo argumento es de tipo T_1 y cuyo resultado es de tipo T_2 (en el lenguaje, este hecho se expresa mediante la *signatura* $f :: T_1 \rightarrow T_2$). Para simular funciones con más de un argumento, simplemente se entiende que la función definida se aplica al primero de ellos y el resultado es una función que se aplica al segundo, y así sucesivamente. Es decir, la expresión $f\ x_1\ x_2\ \dots\ x_n$ se interpreta como $(\dots((f\ x_1)\ x_2)\dots)\ x_n$. Si T_1 es el tipo de x_1 , T_2 el de x_2, \dots, T_n es el tipo de x_n y T es el tipo del resultado, el tipo de f es:

$$T_1 \rightarrow (T_2 \rightarrow (\dots (T_n \rightarrow T))) \quad \text{abreviado como} \quad T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T$$

Esta notación para funciones se denomina *currificada* en honor al matemático Haskell B. Curry que la usó extensivamente en sus trabajos sobre lógica combinatoria durante los años 50¹. La notación currificada permite la creación de varias funciones mediante una sola definición. Considerese la siguiente ecuación:

```
sumar x y = x + y
```

Suponiendo que el operador `+` suma enteros, la ecuación define una función `sumar :: Int -> (Int -> Int)` que, abreviadamente, puede expresarse como `sumar :: Int -> Int -> Int`. La expresión `sumar 3 4` devuelve obviamente 7, pero en su evaluación intervienen dos funciones, a saber: la función `sumar` que, aplicada a 3, devuelve una función sin nombre que podemos llamar `sumar 3`, y esta segunda función que, aplicada a 4, devuelve 7. De hecho, `sumar 3` es una expresión válida, cuyo tipo es `Int -> Int`, que puede ser utilizada en contextos adecuados a este tipo.

Abundando en esta idea, un operador infijo se puede aplicar parcialmente a uno de sus argumentos, dando como resultado una función de un parámetro. Tal construcción recibe el nombre de *sección*. Algunos ejemplos de secciones son los siguientes:

sección	tipo	función
<code>(2*)</code>	<code>Int -> Int</code>	Multiplica por dos un entero
<code>(1+)</code>	<code>Int -> Int</code>	Suma uno a un entero
<code>(2^)</code>	<code>Int -> Int</code>	Dado n , calcula la potencia n de 2
<code>(^2)</code>	<code>Int -> Int</code>	Eleva al cuadrado
<code>(: ['z'])</code>	<code>Char -> [Char]</code>	Crea una lista con el carácter dado y la letra 'z'

Otra posibilidad para crear expresiones de tipo funcional son las llamadas *funciones anónimas*. En ellas se declara el parámetro y el cuerpo de la función, pero no se les asigna un identificador que permita invocarlas en otras partes del programa. Por ejemplo, `\x -> 2*x` es una función anónima de tipo `Int -> Int` que duplica su parámetro. Se puede aplicar a un valor, como en la expresión `(\x -> 2*x) 3` cuya evaluación daría 6. La sintaxis de las funciones anónimas está inspirada en el cálculo lambda. En este formalismo, la función precedente se expresaría como $\lambda x. 2 * x$.

¹Los trabajos de Curry no estaban dirigidos a la programación funcional, pero han aportado importantes fundamentos teóricos a la misma, de ahí el haber tomado su nombre de pila para bautizar al más reciente de los lenguajes funcionales.

Existen muchas funciones de orden superior definidas en el prelude standard, gran parte de ellas para la manipulación de listas, que facilitan notablemente la tarea del programador. Una función de orden superior puede verse como un esquema genérico que encierra un tratamiento frecuentemente utilizado. Por ejemplo, la función `map` aplica una función f a cada elemento de una lista. Genéricamente,

$$\text{map } f [x_1, \dots, x_n] \rightsquigarrow [f x_1, \dots, f x_n]$$

donde el símbolo \rightsquigarrow ha de leerse “se evalúa y da como resultado”. La definición recursiva de `map` se da en la sección 2.3. La función `map` es aplicable siempre que se necesite realizar un mismo tratamiento a todos los elementos de una lista y se desee mantener la estructura de lista. Ejemplos de aplicación de `map` son los siguientes:

función	tipo	significado
<code>map (sumar 3)</code>	<code>[Int]->[Int]</code>	incrementa en 3 cada entero de la lista
<code>map (\x->x-1)</code>	<code>[Int]->[Int]</code>	decrementa en 1 cada entero de la lista
<code>map (==3)</code>	<code>[Int]->[Bool]</code>	deja <code>True</code> donde había un 3, si no deja <code>False</code>
<code>map (: [])</code>	<code>[a]->[[a]]</code>	convierte cada elemento en una lista unitaria

Nótese que las listas origen y destino pueden ser de cualquier tipo. Dichos tipos han de ser consistentes con el de la función f a la que se aplica `map`. En particular, el último ejemplo admite como origen listas de cualquier tipo denotadas por el tipo `[a]`. Estas cuestiones se analizan en detalle en la sección 2.3.

La función de orden superior `foldr` es una de las más importantes para el tratamiento de listas. Tiene tres parámetros: una función binaria $f :: a \rightarrow b \rightarrow b$ una constante $z :: b$ y una lista $xs :: [a]$ donde a y b representan tipos cualesquiera no necesariamente iguales. El resultado de la expresión `foldr f z xs` es un valor de tipo b . La función `foldr` es apropiada siempre que se pretenda operar los elementos de la lista para obtener un valor de cualquier tipo. Por ejemplo, cuando desee obtener la suma de sus elementos, su producto, el *and* lógico de todos ellos y otros resultados semejantes. La mejor forma de comprender como opera `foldr` es recordar cómo se construye una lista $[x_1, \dots, x_n]$:

$$x_1 : (x_2 : (\dots (x_n : []) \dots))$$

y considerar que `foldr` reemplaza la lista vacía `[]` por la constante z y el operador binario $:$ (a veces llamado *cons* por ser una operación constructora de listas) por la versión infija de f , es decir:

$$\text{foldr } f z [x_1, \dots, x_n] \rightsquigarrow x_1 \text{ 'f' } (x_2 \text{ 'f' } (\dots (x_n \text{ 'f' } z) \dots))$$

En términos de `foldr` se pueden definir muchas funciones útiles para el tratamiento de listas:

expresión	tipo	significado
<code>foldr (+) 0</code>	<code>[Int]->Int</code>	suma los elementos de una lista
<code>foldr (*) 1</code>	<code>[Int]->Int</code>	multiplica los elementos de una lista
<code>foldr (&&) True</code>	<code>[Bool]->Bool</code>	realiza el <i>and</i> de una lista de booleanos
<code>foldr (++) []</code>	<code>[[a]]->[a]</code>	concatena una lista de listas

donde el operador binario `++` concatena dos listas.

Matemáticamente hablando, la función `foldr` es un homomorfismo sobre listas en el que las funciones f y z son las homólogas de las constructoras $:$ y `[]`. Cualquier función que sea un homomorfismo del tipo `[a]` al tipo b puede expresarse en términos de `foldr`. Tan sólo hay que encontrar las funciones homólogas f y z (véase [GNPP94] para una discusión más detallada del tema).

Otras funciones de orden superior predefinidas en Haskell son:

llamada	significado
<code>filter p xs</code>	filtra los elementos de la lista xs que satisfagan p
<code>all p xs</code>	devuelve <code>True</code> si todos los elementos de xs satisfacen p
<code>any p xs</code>	devuelve <code>True</code> si algún elemento de xs satisface p
<code>zipWith f xs ys</code>	Similar a <code>map</code> pero con dos listas de entrada (ver sección 2.4)

El programador puede definir adicionalmente sus propias funciones de orden superior. En especial, cuando crea un tipo de datos complejo, como por ejemplo los árboles binarios u otros semejantes, es muy recomendable dotarle enseguida de funciones de orden superior, similares a las funciones `map`, `foldr`, `all p`, `any p`, etc. definidas para las listas, que encapsulen los tratamientos más frecuentes para ese tipo de datos. De ese modo evitará una gran cantidad de diseño redundante y, lo que es más importante, una cantidad no despreciable de errores.

2.3 Sistema de tipos

2.3.1 Polimorfismo, sobrecarga e inferencia automática

Utilizando palabras de Luca Cardelli [Car89], Haskell es un lenguaje “lleno de tipos” aunque a primera vista pudiera parecer lo contrario. Cada expresión tiene un tipo bien definido y el lenguaje comprueba que los identificadores se combinan correctamente de acuerdo con sus tipos. Es decir, Haskell es un lenguaje *con disciplina de tipos*. Sin embargo, a diferencia de los lenguajes imperativos con disciplina de tipos, no es necesario declarar explícitamente el tipo de los objetos que se definen. El lenguaje posee un algoritmo de inferencia automática de tipos que le permite deducir el tipo de los objetos, tanto en el momento de su definición como en el de su uso. Si el tipo de algún uso no es compatible con el tipo inferido en la definición, se produce un error. Todas las comprobaciones de tipos se realizan en tiempo de compilación.

Por ejemplo, para la función `triplo` definida en la sección 2.1 Haskell deduce el tipo `triplo: Int -> Int` debido a la aparición de la constante entera `3` como uno de los argumentos del operador `*`. Ello fuerza a que `x` y el resultado hayan de ser de tipo `Int`. Por un algoritmo similar deduciría de su definición que el tipo de la función `suma` de la sección 2.1 es `[Int] -> Int`.

Estos tipos reciben el nombre de tipos concretos o *monomórficos*. En Haskell existen otras dos clases de tipos: los tipos *polimórficos* y los tipos *sobrecargados*. En ambos casos las expresiones de tipo correspondientes contienen *variables de tipo* a las que se designa mediante identificadores (con minúscula) no utilizados para otro propósito, p.e. `a`, `b`, `c`, etc.

En los tipos polimórficos, las variables de tipo se suponen cuantificadas universalmente. En definitiva, un tipo polimórfico define una familia de infinitos tipos monomórficos, uno para cada sustitución posible de las variables de tipo. Diremos que un tipo T_2 es una *concreción* de un tipo polimórfico T_1 si T_2 puede obtenerse de T_1 sustituyendo sus variables por expresiones de tipo. Por ejemplo, dada la siguiente definición de la función `map` introducida en la sección 2.2,

```
map f []      = []
map f (x:xs) = f x : map f xs
```

Haskell deduce el tipo `map :: (a -> b) -> [a] -> [b]`, donde `a` y `b` son variables de tipo. Es obvio que el segundo parámetro de `map` y el resultado de la llamada son de tipo lista. Que el primer parámetro es una función se deduce de la expresión `f x` cabeza de la lista resultado de la segunda ecuación. Al no tener más información sobre `x`, le asigna el tipo polimórfico `a`, y al carecer de otras restricciones sobre `f`, decide que sus resultados son de tipo `b`. Eso completa la deducción. El tipo deducido es el más general posible y se puede demostrar que, caso de existir, el tipo más general de una expresión es único. Los usos que se hace de `map` más adelante son compatibles con este tipo más general. Por ejemplo, en la expresión `map (==3)` la función que toma el papel de `f` es `(==3) :: Int -> Bool`, que es una concreción de `a -> b` en la que `a` se reemplaza por `Int` y `b` por `Bool`. Sustituyendo estos valores en el resto de la expresión de tipos, se obtiene para la expresión `map (==3)` el tipo `[Int] -> [Bool]`.

La intuición subyacente a los tipos polimórficos es la de que las funciones definidas sobre ellos tienen un comportamiento idéntico cualquiera que sea la concreción del tipo sobre la que se aplique. Un ejemplo ilustrativo es el de la longitud de una lista: la función correspondiente se llama `length :: [a] -> Int`. El algoritmo que calcula la longitud de una lista es el mismo independientemente de que los elementos de la lista sean enteros, booleanos, o de cualquier otro tipo. Así, el polimorfismo constituye un importante factor de abstracción que permite encapsular en una definición muchas funciones a la vez (compárese esta situación con la de un lenguaje como Pascal en la que una pequeña variación tal como modificar el tipo de los elementos de un *array*, obliga normalmente a reprogramar el algoritmo para introducir tan sólo ligeras variaciones).

La tercera categoría de tipos son los sobrecargados, también llamados a veces tipos con polimorfismo *ad hoc*. Hay indicios de sobrecarga en muchos lenguajes. Por ejemplo, en Pascal, se emplea el símbolo `+` para designar a la vez la suma de enteros y la suma de números en coma flotante. Se dice en estos casos que el símbolo está *sobrecargado*, queriéndose indicar que se está empleando el mismo nombre para dos algoritmos *distintos*.

Haskell también tiene operaciones sobrecargadas (el propio operador `+` es un ejemplo), pero además permite al programador crear las suyas propias. Esta facilidad es compatible con la existencia de los tipos polimórficos y con el mecanismo de inferencia automática arriba descritos². Haskell es el primer lenguaje

²Ocasionalmente pueden producirse situaciones de ambigüedad en el tipo deducido que han de ser desambiguadas por

que proporciona tal sistema de tipos y, de hecho, es la característica que más distingue a Haskell de los lenguajes funcionales previos a él.

La expresión de un tipo sobrecargado también tiene variables de tipo pero, a diferencia de lo que sucede en los tipos polimórficos, aquí las variables no están cuantificadas universalmente, sino que han de tomar valores en una o más *clases* de tipos previamente definidas. Por ejemplo, existe la clase de los tipos numéricos, denominada la clase `Num`, a la que pertenecen diversas categorías de enteros, racionales, números en coma flotante y números complejos. Todos ellos tienen definidas las operaciones `+`, `*`, `-` y otras. Para expresar que estas operaciones tienen un tipo sobrecargado, se escribe:

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

cuya interpretación es: “el tipo de las tres operaciones es de la forma `a -> a -> a` para todo tipo `a` que pertenezca a la clase `Num`”. La expresión `(Num a)` que restringe los posibles valores de `a` se denomina *contexto* y, en el caso general, puede nombrar varias clases.

Otra clase interesante es la clase `Eq` de los tipos con igualdad (y desigualdad). Las funciones `(==)`, `(/=)` `:: a -> a -> Bool` tienen sentido para la mayoría de los tipos de Haskell pero no para todos. Por ejemplo, no existe un algoritmo efectivo para comparar si dos funciones son iguales cuando su dominio de definición tiene infinitos valores. Por tanto, lo correcto es considerar la operación `==` como sobrecargada (y no como polimórfica, tal como sucedía en algunos lenguajes previos), si bien en este caso la clase consta de infinitos tipos.

El mecanismo de inferencia de tipos tiene en cuenta las restricciones que la sobrecarga impone a las variables de tipo, y propaga esas restricciones adecuadamente. Así, para la función `insert` de la sección 2.1 deduciría el siguiente tipo:

```
insert :: (Ord a, Eq a) => a -> [a] -> [a]
```

donde `Ord` es la clase de los tipos que tienen definida una relación de orden. El contexto `(Ord a, Eq a)` proviene de haber utilizado las operaciones `<` e `==` en la definición de `insert`.

Haskell predefine una docena de clases y establece una relación de inclusión entre algunas de ellas (por ejemplo, la clase `Ord` es subclase de la clase `Eq`). Más interesante aún es el hecho de que el programador puede crear nuevas clases o ampliar algunas de las existentes con nuevos tipos. Por ejemplo, podría definir los árboles binarios y dotarles de una relación de igualdad. Mediante una declaración apropiada, puede sobrecargar el símbolo `(==)` con el significado de igualdad entre árboles binarios e incluir este tipo en la clase `Eq`.

Por último, Haskell no obliga, como se ha dicho, a declarar explícitamente los tipos de los objetos definidos, pero sí permite hacerlo. Para ello basta incluir una declaración de la forma *identificador* `:: expresión_de_tipo`. Existen algunas razones por las que, en ciertos casos, puede ser adecuado escribir tales declaraciones:

- Por legibilidad y autodocumentación del texto, especialmente cuando se trata de objetos que se exportan de un módulo.
- Para restringir el uso de ciertos objetos. A veces, el programador desea que una definición se use con tipos más particulares que los que permitiría el tipo más general inferido de la definición. Por ejemplo, ante una definición de la forma

```
id x = x
```

el tipo inferido para la función `id` es `id :: a -> a`. Si el programador desea que la función `id` sólo se aplique a listas, podría incluir una declaración de la forma `id :: [a] -> [a]`. Esta declaración es compatible con el tipo inferido y prevalece sobre aquél.

- Para desambiguar ciertas expresiones a las que, debido a la sobrecarga, el algoritmo de inferencia no puede asignar un tipo único.

el programador.

2.3.2 Tipos predefinidos

Haskell suministra una rica variedad de tipos predefinidos como corresponde a un lenguaje de propósito general. Los tipos atómicos incluyen los siguientes:

- Dos variedades de enteros: el tipo `Int`, ligado a la arquitectura del computador subyacente y, por tanto, acotado, y el tipo `Integer`, no acotado, que representa genuinamente los enteros de las matemáticas. La utilidad del segundo, menos eficiente, se manifestará obviamente en algoritmos que trabajen con grandes números. Los dos tipos pertenecen a una misma clase denominada `Integral`.
- Dos variedades de números en coma flotante: `Float` para precisión simple y `Double` para doble. Los dos se agrupan en una clase común denominada `RealFloat`.
- Un tipo sobrecargado `Ratio` a que construye números racionales a partir de pares de números de la clase `Integral`. Por tanto, podrán construirse racionales con enteros acotados de tipo `Int` o con enteros no acotados de tipo `Integer`.
- Un tipo sobrecargado `Complex` a que construye números complejos a partir de pares de números de la clase `RealFloat`. Por tanto, existen complejos de simple y de doble precisión.
- El tipo `Char` de los caracteres, que se implementan sobre la codificación ASCII.
- El tipo `Bool` de los booleanos.

Todos ellos vienen dotados de un amplio conjunto de operaciones, la mayor parte de las cuales están sobrecargadas y, por tanto, se agrupan de modo natural en clases. La estructura de las mismas es algo compleja para detallarla aquí. A modo de ejemplo, diremos que los operadores `(+)`, `(-)` y `(*)` son aplicables a todos los números (clase `Num`), en cambio, el operador `(/)` sólo lo es a los racionales, reales y complejos (clase `Fractional` común a todos ellos) y las funciones trigonométricas tales como `sin`, `cos`, `tan`, `asin`, etc., sólo están definidas para los tipos `Float`, `Double` y `Complex` a (agrupados en la clase `Floating`). Asimismo, las constantes enteras `-1, 0, 1, 2, ...` están sobrecargadas para los tipos `Int` y `Integer` y las constantes en coma flotante lo están para los tipos `Float` y `Double`.

Las constantes de tipo carácter se escriben entre comillas simples (p.e. `'a'`, `'A'`) y existe una sintaxis especial para escribir caracteres no imprimibles (p.e. `\LF` es el carácter fin de línea). Las cadenas de caracteres (tipo `String`) se consideran listas, lo que aporta la ventaja de poder usar todas las funciones predefinidas para las mismas. No obstante, se permite usar una sintaxis especial para las constantes. Así, `"Hola"` es equivalente a `['H', 'o', 'l', 'a']`.

Finalmente, los booleanos están provistos de las operaciones habituales: `&&` (*and*), `||` (*or*) y `not` (negación).

En cuanto a los tipos estructurados, se suministran las listas, cuya sintaxis ha ido apareciendo a lo largo de este trabajo, las tuplas y los *arrays*. Una tupla es una secuencia finita de valores de tipos no necesariamente iguales. Por ejemplo, `(3, True, "Hola")` es una tupla de tres valores cuyo tipo es `(Int, Bool, [Char])`. Distinto número de elementos implica tipos distintos. Por tanto, existen infinitos tipos tupla diferentes. Existe incluso el tipo `()` (tupla vacía) que contiene un único valor, denotado también `()`. Los componentes de una tupla se extraen normalmente por posición mediante encaje de patrones. Las tuplas son especialmente útiles cuando se necesita definir funciones que devuelven más de un resultado.

Las listas tienen como constructores básicos la lista vacía `[] :: [a]` y el constructor infijo `cons`, denotado `(:) :: a -> [a] -> [a]`, que añade un elemento a la cabeza de una lista y asocia hacia la derecha. Los lenguajes funcionales han desarrollado una rica sintaxis para facilitar la programación con listas. Así, `[1, 2, 3]` es una abreviatura de `1:2:3:[]`, y `[1..10]` es la lista de los números de 1 a 10. En Haskell se pueden definir listas conceptualmente infinitas gracias a la técnica de evaluación perezosa (ver apartado 2.4). Por ejemplo, la lista de todos los naturales se escribe `[0..]` y la de los números impares `[1, 3..]`. Pero la aportación más notable de los lenguajes funcionales en cuanto a sintaxis compacta para listas es la llamada notación ZF³ o definición de listas por comprensión⁴. Una definición escrita en notación ZF tiene el siguiente aspecto:

$$[\textit{expresión} \mid \textit{cualif}_1, \dots, \textit{cualif}_n]$$

³Inspirada en la notación conjuntista de los matemáticos Zermelo y Fränkel.

⁴La palabra comprensión se emplea aquí como término opuesto a extensión.

donde cada cualificador *qualif_i* puede ser un *generador* o un *filtro*. En el primer caso tiene la sintaxis *patrón* `<- lista` y se utiliza para asignar distintos valores a las variables del patrón. Dichas variables intervienen en la *expresión* a la izquierda del símbolo `|`. Los filtros son expresiones booleanas que se utilizan para dejar en la lista final sólo aquellos valores generados que satisfagan las mismas. Las siguientes expresiones definen la lista de los divisores de un número y la lista (infinita) de los números primos:

```
divisores n = [d | d <- [1..n], n `mod` d == 0]
primos      = [p | p <- [2..], divisores p == [1,p]]
```

Si existen varios generadores, los de la derecha varían más rápido que los de la izquierda y pueden hacer uso de las variables generadas por los patrones precedentes. Así, la expresión

```
[(x,y) | x <- [1..3], y <- [1..x]]
```

genera la lista `[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]`.

El último tipo estructurado predefinido en Haskell es el tipo *array*. Ha habido mucha controversia acerca de la inclusión o no de *arrays* en los lenguajes funcionales. Todos los lenguajes imperativos los incluyen, por ser el tipo que mejor se adapta a la arquitectura de la memoria RAM subyacente y el que permite sacar partido al acceso en tiempo constante a cualquier posición de la misma. Sin embargo, parece estar íntimamente ligado a la noción de estado, noción que no existe en los lenguajes funcionales. Por ejemplo, la función `insert` de la sección 2.1 que inserta un elemento en una lista, conceptualmente trabaja con dos listas distintas: la de entrada y la de salida. Una implementación ingénuo generaría una nueva lista que sería prácticamente una copia de la que se suministra como parámetro. Si ésta no es requerida en ninguna otra subexpresión, simplemente se descartaría y el algoritmo de *recolección de desperdicios* rescataría en su momento la memoria que ocupaba.

En el caso de los *arrays*, cualquier modificación implicaría la copia del mismo, con lo que se perderían las ventajas del acceso en tiempo constante. Los *arrays* han sido incluidos finalmente en Haskell, confiando a las técnicas de análisis estático del texto —las cuales se han desarrollado notablemente en los últimos años— la misión de decidir en qué casos es necesario generar una copia. Por supuesto, una programación poco cuidadosa con *arrays* puede hacer desaparecer sus potenciales ventajas. El uso recomendado es la creación *monolítica* (es decir, crear la matriz suministrando de golpe todo su contenido) y el acceso posterior para consulta. También se suministra la modificación *incremental*, es decir, la modificación de una o varias posiciones de la misma. En ese caso, no se genera copia si el valor de la matriz previo a la modificación no es requerido en ninguna subexpresión. Para más detalles, véase [HPJW92, HF92].

2.3.3 Tipos definidos por el programador

Todo lenguaje con pretensiones de servir para la programación en gran escala ha de permitir al programador definir sus propios tipos de datos. En Haskell hay dos modos de hacerlo: renombrando tipos previamente definidos o construyéndolos explícitamente. En el primer caso se emplea la sintaxis:

```
type Nombre a1 a2 ... an = expresión_de_tipos
```

donde $n \geq 0$. Las siguientes definiciones dan algunos ejemplos de renombramientos:

```
type String = [Char]
type Racional = (Int,Int)
type Pila a = [a]
```

Los renombramientos se utilizan fundamentalmente por legibilidad. Por ejemplo, es más ilustrativo declarar `sumaRacionales :: Racional -> Racional -> Racional` que utilizar `(Int,Int)` directamente. El lenguaje trata los renombramientos como tipos idénticos a los originales de los cuales proceden. De hecho, una vez realizada la traducción, sólo utiliza los originales. Es decir, no distingue internamente entre ambos, por lo que no se debe esperar protección contra usos indebidos. Por ejemplo, podría concatenarse un objeto de tipo `Pila Int` con otro de tipo `[Int]` sin que existiera violación de tipos.

El segundo modo de definir tipos es construir su conjunto de valores explícitamente. Estos tipos se denominan *algebraicos* y la sintaxis para definirlos es lo suficientemente versátil como para definir tipos que en otros lenguajes necesitan de más de un mecanismo. Mediante el mecanismo de tipos algebraicos se pueden definir los siguientes tipos:

enumerativos Formados por un dominio finito de valores, cada uno de los cuales recibe un nombre distinto.

coerciones Un tipo se define aplicando una coerción a otro previo. Una coerción es una aplicación inyectiva de un conjunto de valores en otro.

uniones disjuntas Un tipo se define mediante la unión disjunta de dos o más tipos.

inductivamente generados Los valores del tipo se construyen aplicando operaciones constructoras a valores previamente generados. En los lenguajes imperativos estos tipos se construyen normalmente utilizando punteros.

La sintaxis general para definir un tipo algebraico es la siguiente:

$$\text{data Nombre } a_1 a_2 \dots a_m = K_1 e_{11} \dots e_{1n_1} | \dots | K_r e_{r1} \dots e_{rn_r}$$

donde las $a_i, i \in \{1..m\}, m \geq 0$ son variables de tipo, las $K_j, j \in \{1..r\}, r \geq 1$ se denominan *constructoras* del tipo *Nombre*, y son identificadores distintos escritos comenzando por mayúscula, y las $e_{jk}, j \in \{1..r\}, k \in \{1..n_j\}, n_j \geq 0$ para todo j , son expresiones de tipo en las que pueden aparecer las variables de tipo a_i . Está permitido que el propio tipo definido, es decir, la expresión de tipos *Nombre* $a_1 \dots a_m$, aparezca como subexpresión de algunas de las expresiones e_{jk} . Veamos algunos ejemplos:

<code>data Color = Rojo Verde Azul</code>	Enumerativo
<code>data Bool = True False</code>	Enumerativo
<code>data Tupla a b = Tup a b</code>	Coerción
<code>data Fruta = Pera Int Manzana Int</code>	Unión
<code>data Cola a = VerComoCola [a]</code>	Coerción
<code>data Pila a = Pvacia Apilar (Pila a) a</code>	Inductivamente generado

El papel de las constructoras es doble: por un lado sirven para generar y dar nombre a los valores del tipo. Así, `True` y `False` son los dos únicos valores de tipo `Bool`, `Pera 3` y `Manzana 4` son dos valores de tipo `Fruta`, `VerComoCola [1,2,3]` es un valor de tipo `Cola Int`, y `Apilar (Apilar Pvacia 1) 2` es un valor de tipo `Pila Int`; por otro, sirven para formar patrones de los valores del tipo y con ellos definir funciones. Los siguientes son ejemplos de funciones definidas mediante patrones (se adjunta el tipo inferido para cada una):

```
(||)::Bool -> Bool -> Bool
True  || _ = True
False || x = x

(+)::Fruta -> Fruta -> Fruta
(Manzana x) + (Manzana y) = Manzana (x+y)
(Pera x) + (Pera y) = Pera (x+y)

longitud::Cola a -> Int
longitud (VerComoCola xs) = length xs

cima::Pila a -> a
cima (Apilar p x) = x
```

Notese que tanto `(+)` como `cima` son funciones parciales. Las expresiones de la forma `Pera x + Manzana y` o `cima Pvacia` no están definidas. Cualquier intento de evaluarlas conduciría a un error en tiempo de ejecución.

La facilidad de tipos algebraicos permite definir tipos de datos que admiten constructoras libres, es decir, aquellos que son isomorfos al álgebra de términos libremente generada por las constructoras. Cada término sintácticamente distinto denota un valor distinto del tipo. Desafortunadamente, existen multitud de tipos de datos interesantes que no admiten constructoras libres. Los ejemplos clásicos son los conjuntos y los multiconjuntos, para los cuales las constructoras habituales, p.e. `Cvacio` y `Añadir`, satisfacen leyes conmutativas o de otro tipo. Así, los términos

`Añadir (Añadir Cvacio 1) 2` y `Añadir (Añadir Cvacio 2) 1`

son sintácticamente distintos y, por tanto, representan valores distintos, si bien nos gustaría que denotaran el mismo valor. Ningún lenguaje funcional permite expresar leyes algebraicas entre términos formados por constructoras, aunque ha habido algunos intentos en ese sentido que finalmente han fracasado. La razón principal de no admitir estas leyes es que, en general, desaparecen las formas normales de los valores o éstas se convierten en clases de términos. En el caso general de ecuaciones cualesquiera, no es decidible si dos términos pertenecen o no a la misma clase.

Por ello, los lenguajes funcionales ofrecen un tercer mecanismo de definición de tipos que permite simular el comportamiento de los tipos no libres: el mecanismo de *tipos abstractos*. Éste es similar al que existe en algunos lenguajes imperativos. Un tipo abstracto tiene una interfaz visible y una representación oculta en términos de otros tipos. Las operaciones visibles proporcionan el comportamiento deseado por el usuario si bien están implementadas en términos de operaciones sobre los tipos de la representación. Por ejemplo, podemos representar los conjuntos por listas ordenadas sin repeticiones. En Haskell ésto se expresaría así:

```
module Conjuntos (Conj, cvacio, añadir, pertenece)
where data Conj a = Mcc [a]
      cvacio = Mcc []
      añadir (Mcc xs) x = Mcc (insert x xs)
                        where insert ... -- ver sección 2.1
      pertenece (Mcc []) x = False
      pertenece (Mcc (y:ys)) x
        | x < y = False
        | otherwise = x==y || pertenece (Mcc ys) x
```

El mecanismo de módulos de Haskell sirve, entre otros propósitos, para definir tipos abstractos. De un módulo sólo es visible la interfaz, que es la lista de identificadores entre paréntesis detrás del nombre del módulo. En este caso, por tanto, es visible el tipo `Conj` pero no su representación. La constructora `Mcc` establece una coerción entre listas y conjuntos, pero no es visible fuera del módulo. Los usuarios sólo pueden construir conjuntos mediante las funciones (no confundir con constructoras) `cvacio` y `añadir`, y pueden consultarlos mediante la operación `pertenece`.

2.4 Evaluación perezosa

A partir de cierto momento en la historia de los lenguajes funcionales (que podemos situar en la aparición del lenguaje Miranda [Tur85]), se ha reconocido la superioridad de la evaluación perezosa de expresiones frente a su alternativa, la evaluación impaciente. Los lenguajes más recientes son *perezosos*, también llamados *no-estrictos*. La diferencia entre ambas es la siguiente: en la evaluación impaciente, una aplicación de función de la forma $f e_1 e_2 \dots e_n$ se ejecuta reduciendo primero las expresiones e_i a forma normal y después aplicando f al resultado. Este orden de evaluación corresponde al concepto de paso de parámetros por valor existente en la mayoría de los lenguajes imperativos. En la evaluación perezosa, las e_i no son evaluadas, o lo son tan sólo parcialmente, si no son realmente requeridas para la evaluación de f . Considérese la definición de la función que indica si una lista es o no vacía:

```
vacia [] = True
vacia (x:xs) = False
```

y la expresión `vacia [x+y | x <- [1..10], y <- [1..x]]`. Con evaluación impaciente, primero se genera la lista y luego se aplica la operación `vacia`. Con evaluación perezosa, la lista se evalúa hasta que se puede aplicar una de las ecuaciones de `vacia`. En este caso, hasta que la lista ofrece aproximadamente el siguiente aspecto: `(1+1):[x+y | x <- [2..10], y <- [1..x]]`. En este punto ya puede devolver `False` con lo que el resto de la lista no es evaluada. Ni siquiera ha sido necesario reducir la cabeza de la lista (es decir, realizar la operación `1+1`).

Además de las ventajas obvias de eficiencia⁵ la evaluación perezosa es siempre capaz de encontrar la forma normal de una expresión, si ésta existe, lo que no sucede con la evaluación impaciente. Por tanto, un lenguaje estricto podría no terminar en evaluaciones que sí terminarían con uno no estricto. Considérese el siguiente ejemplo sencillo:

⁵La comparación de la eficiencia de las estrategias impaciente y perezosa es siempre difícil de realizar. Si bien la estrategia perezosa puede ahorrar la evaluación de muchas subexpresiones, si la forma normal de una expresión es finalmente requerida, la evaluación impaciente de la misma es menos costosa de implementar.

```

const x y = x
bucle x   = bucle x

```

La expresión `const 1` (`bucle 2`) es no terminante en un lenguaje impaciente aunque su forma normal existe y vale 1. Éste sería el resultado alcanzado por un lenguaje perezoso.

La estrategia perezosa permite la definición de estructuras de datos conceptualmente infinitas lo que conduce a un modo de programar más compacto e intuitivo. La única precaución que ha de tener el programador es impedir que una expresión requiera la evaluación completa del objeto infinito. La siguiente ecuación define la lista de las infinitas potencias de 10:

```

pot10 = 1 : map (*10) pot10

```

Por ejemplo, la expresión `length pot10` conduciría a no terminación. En cambio `take 5 pot10`, donde `take` es una función predefinida que extrae los primeros n elementos de una lista, produce la lista `[1,10,100,1000,10000]`.

La superioridad de la evaluación perezosa se manifiesta en el hecho de que el programador puede desentenderse en gran medida del control y expresar sus funciones de un modo más intuitivo, sabiendo que el mecanismo de evaluación “trabaja” lo menos posible. Considérese la definición de la función que indica si un elemento pertenece o no a una lista:

```

pertenece x []      = False
pertenece x (y:ys) = x==y || pertenece x ys

```

Aparentemente, la función compara el valor de x con todos los elementos de la lista y luego realiza el *or* de los resultados de tales comparaciones. Sin embargo, la evaluación perezosa del operador `||` (véase su definición en el apartado 2.3.3) hace que pueda devolver `True` en el momento en que el parámetro izquierdo se reduce a `True`. La consecuencia es que, cuando se encuentra un elemento de la lista que coincide con x , no es necesario aplicar la función `pertenece` al resto de la lista.

La combinación de orden superior y evaluación perezosa permite un estilo de programación que se reduce en muchos casos a ensamblar componentes. El uso de recursión explícita (equivalente en cierto modo a la iteración en la programación imperativa) obliga a un mayor esfuerzo de diseño por parte del programador y aumenta la probabilidad de cometer errores. Por ello, disponer de conceptos que, en muchos casos, permitan evitarla es algo que debe ser bien recibido. Como muestra de estas afirmaciones, considérese la siguiente función que convierte enteros de ASCII a binario, es decir, `convertir "123" ~> 123`:

```

convertir :: [Char] -> Int
convertir cs = sum (zipWith (*) pot10 (reverse cs'))
               where cs' = map (\ c -> ord c - ord '0') cs

```

Las funciones predefinidas `sum` y `reverse` respectivamente suman e invierten una lista, la función `ord` devuelve el entero que corresponde a un carácter en el código ASCII, y la función de orden superior `zipWith` está definida en el preludio `standard` de Haskell como sigue:

```

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _     _     = []

```

Obsérvese que, si las listas son de diferente longitud, la función `zipWith` descarta el resto de la lista más larga. Esto garantiza que la lista infinita `pot10` sólo es evaluada hasta conseguir tantas potencias de 10 como cifras tiene la lista `cs`. Aquí, de nuevo, el programador puede desentenderse del control de la terminación del algoritmo. En una programación impaciente, o en una versión imperativa del algoritmo, sería necesario contar primero las cifras de `cs` y generar exactamente ese número de potencias de 10.

3 Pros y contras de la programación funcional

Una vez presentadas las características más relevantes de la programación funcional (por brevedad, hemos dejado de lado algunos aspectos también merecedores de consideración tales como la definición de clases, el sistema de módulos, la entrada/salida y la librería de funciones predefinidas), en un lenguaje que representa el estado del arte, en esta sección se exponen (subjétivamente) argumentos a favor de la

programación funcional y se responde a algunas de las críticas más frecuentes al paradigma. Por claridad, se han organizado los distintos argumentos en apartados separados. Los acrónimos PF y LF se emplearán respectivamente como abreviaturas de Programación (o Programa) Funcional y de Lenguaje Funcional.

Simplicidad

El mejor argumento en favor de la PF es que el modelo de cómputo y los lenguajes derivados del mismo son conceptualmente más simples que sus contrapartidas imperativas. Un dato que apoya esta afirmación es el volumen de los documentos que describen unos lenguajes y otros. El manual de referencia de Haskell [HPJW92], excluyendo los preludios, ocupa 80 páginas y el número de identificadores reservados es de 23. Un lenguaje que representa el estado del arte en programación imperativa como Ada [Bar84] tiene un manual de referencia [oD81] de más de 200 páginas y el número de identificadores reservados que utiliza es de 62.

La razón de esta mayor simplicidad es que un LF consta de muy pocos elementos y casi no tiene excepciones a la hora de combinarlos. En esencia, un programa funcional es un conjunto de *expresiones* que se transforman unas en otras de acuerdo a las reglas del cálculo ecuacional. Al estar ausente la noción de estado, se eliminan multitud de conceptos ligados a él tales como la modificación de variables, el control de la secuencia exacta en que se suceden los cambios o complicados mecanismos de paso de parámetros.

Posibilidad de razonamiento formal

Sobre un programa imperativo se puede razonar formalmente, pero cuesta un esfuerzo nada despreciable (de ello son buenos conocedores los alumnos de los primeros cursos de Informática). Para razonar sobre un programa imperativo, primero se ha de expresar su comportamiento pretendido en una lógica externa al lenguaje (la lógica de predicados), y a continuación se ha de razonar en ese mundo más ordenado. La causa de esta dificultad hay que buscarla de nuevo en la existencia de un estado. Más precisamente, en el hecho de que una misma variable denota valores distintos en partes distintas del texto, o en el mismo lugar del texto en instantes distintos.

La PF goza de la propiedad de *transparencia referencial* que se define como el hecho de que cualquier expresión, y en particular una variable, denota el mismo valor en cualquier parte del programa que esté referida al mismo ámbito de visibilidad de los identificadores. De hecho, fijado un ámbito de visibilidad, no se permite cambiar el valor de un identificador. Las variables tienen así el mismo significado que en matemáticas: o denotan un sólo valor, como en la declaración $\pi = 3.14159$, o están cuantificadas universalmente, como x en la definición `duplo x = 2*x`.

Esta propiedad es la que permite el razonamiento ecuacional, es decir, la sustitución de iguales por iguales. Se puede así razonar sobre un PF y demostrar propiedades tales como la equivalencia entre dos funciones o la satisfacción de leyes asociativas, conmutativas o de cualquier otro tipo, sin salirse de la lógica del propio lenguaje. Igualmente se puede realizar transformaciones de programas que preservan el significado (p.e. con el fin de aumentar la eficiencia), aplicando leyes algebraicas previamente demostradas.

Concisión

Se han hecho experimentos (ver p.e. [HPJ94]) en los que se da un factor de reducción de aproximadamente diez en el número de líneas de código que es necesario escribir para resolver un problema dado al programar en un LF, frente a hacerlo en uno imperativo. Teniendo en cuenta que la productividad de un programador depende en mucha medida del número de líneas que ha de escribir, probar, documentar, mantener, etc., no cabe duda de que el paradigma funcional ofrece, en principio, muy buenas perspectivas de aumento de la productividad. Menor volumen de código significa, a la vez, menor número de errores, con lo que cabe esperar también una mejora en el otro cuello de botella del desarrollo de programas: la fiabilidad.

A veces, concisión significa que el programa adquiere un aspecto críptico (p.e. el lenguaje APL permite programas muy concisos pero a la vez ilegibles). El lector mismo puede juzgar si los programas presentados le parecen o no legibles. Sin duda, se puede programar de forma críptica en un LF, pero la crítica es igualmente aplicable a cualquier lenguaje de programación. La pregunta adecuada es si un LF es *inherentemente* más críptico que un lenguaje imperativo. En el apartado sobre críticas a la PF, responderemos a esta cuestión.

Abstracción

Comparada con la programación imperativa, la PF ha de verse necesariamente como un modo más abstracto de programar. Abstracto se entiende aquí como, a la vez, mayor lejanía al computador y mayor cercanía al modo de razonar de los programadores. La distancia entre los lenguajes imperativos y la arquitectura del computador subyacente, la arquitectura Von Neumann, es mucho más pequeña que en el caso funcional. De hecho, las construcciones de un lenguaje imperativo son un recubrimiento edulcorado de los correspondientes dispositivos del hardware. Así, la memoria está representada por las variables del programa, los saltos condicionales e incondicionales del computador, por las instrucciones de control del lenguaje, y las operaciones de transferencia de datos entre la memoria y la UCP, por la instrucción de asignación.

No es posible establecer este paralelismo en el caso de la PF. Su modelo de cómputo está basado, como se ha dicho, en la lógica ecuacional. De hecho, esta mayor distancia entre el modelo de cómputo de la PF y el computador Von Neumann es uno de los obstáculos para que la PF alcance una eficiencia comparable a la de la programación imperativa.

Además de poseer un modelo de cómputo más cercano al modo de pensar humano, la PF introduce potentes mecanismos adicionales de abstracción como son el polimorfismo, el orden superior y la evaluación perezosa, no presentes en la programación imperativa. En la sección precedente se ha tratado de mostrar sus beneficios. Estos mecanismos proporcionan, en palabras de J. Hughes [Hug90], nuevos tipos de “pegamento” para ensamblar fragmentos de programa. Así, el polimorfismo permite utilizar unos tipos de datos como parámetros para otros sin limitación alguna; las funciones pueden ser pasadas como parámetros unas a otras, también sin limitación, gracias al orden superior; y la evaluación perezosa permite separar los aspectos de corrección de los aspectos de control, haciendo que se puedan ensamblar juntos objetos que de otro modo necesitarían ser reprogramados.

El resultado de todo ello es un grado mayor de reutilización del código y, quizás más importante, un modo distinto de concebir la programación: más como la tarea de componer nuevos rompecabezas con piezas conocidas, que como la de fabricar constantemente piezas nuevas.

Paralelismo potencial

El desarrollo de arquitecturas paralelas con costes progresivamente descendentes pone a los programadores ante la presión de obtener un rendimiento adecuado de las mismas. Desafortunadamente, los progresos en el mundo de la programación no están a la altura de la evolución lograda en el hardware. La programación concurrente sigue siendo hoy una tarea llena de problemas: a las dificultades de la parte imperativa de los lenguajes concurrentes, se añaden las dificultades debidas a la sincronización y comunicación de los procesos, con los riesgos conocidos de potenciales bloqueos, violación de la exclusión mutua, injusticias en el reparto de recursos, etc.

La evaluación de expresiones en PF admite un paralelismo obvio: dos subexpresiones cuya evaluación es realmente requerida, pueden evaluarse independientemente y en paralelo sin riesgo de incorrección debido a la ausencia de efectos laterales, consecuencia a su vez de la transparencia referencial. La creación de procesos y la sincronización y comunicación entre los mismos serían transparentes al programador. El único programa concurrente que habría que construir explícitamente sería el intérprete funcional.

Un programa funcional que necesitase grandes recursos de cálculo podría así extraer un alto rendimiento de una máquina multiprocesador sin que ello supusiera un esfuerzo extra por parte del programador. La situación es muy diferente en el caso de los programas imperativos.

Crítica 1: lenguajes de juguete

Una de las críticas escuchadas durante mucho tiempo acerca del paradigma funcional era que los lenguajes existentes eran meros productos de laboratorio, buenos para investigar, pero en ningún modo adecuados para una programación en gran escala.

Si esta crítica pudo tener alguna justificación a comienzos de los años 80, hoy ha dejado radicalmente de tenerla. Lenguajes como Standard ML [Mil84, HMM86, Wik87], Miranda [Tur85, Tur86] o Haskell [HPJW92, HF92], están provistos de todos los mecanismos y aditamentos necesarios para el desarrollo de grandes programas: potentes tipos de datos numéricos con su librería matemática asociada, módulos y compilación separada, entrada/salida desde/hacia cualquier tipo de periférico o dispositivo de almacenamiento externo, e incluso posibilidad de interfaz con programas desarrollados en otros lenguajes.

Crítica 2: ineficiencia

La principal crítica que se hace a la PF es que no es posible alcanzar con ella la misma eficiencia que con la programación imperativa. Esta crítica es radicalmente correcta. Bajo ciertos supuestos que no tienen por qué ser inmutables (arquitectura Von Neumann y máquina monoprocesador), los lenguajes imperativos son los más aptos para extraer el máximo rendimiento del computador, dado que se adaptan perfectamente a su arquitectura. Los programas funcionales ocupan más memoria y tardan más en ejecutarse⁶.

Conviene recordar en este sentido, que esta misma crítica la sufrieron los primeros lenguajes de alto nivel con respecto a la programación en lenguaje ensamblador. Obviamente, si aquéllos han triunfado finalmente frente a éste es porque se considera que el precio de su ineficiencia relativa está ampliamente compensado con los beneficios que reporta. Estos beneficios resultan ser prácticamente los mismos que hoy ofrece el paradigma funcional frente al imperativo: mayor productividad de los programadores y menor tasa de errores.

Hay que preguntarse entonces, si unos pocos megaoctetos de más y una máquina más rápida, en una época en que el precio del hardware tiene cada vez menos incidencia en el precio final del producto, son o no precios que están justificados a cambio de reducir los costes de desarrollo del software.

Por otra parte, existen multitud de tareas de programación en las que el tiempo de ejecución no es especialmente crítico y en cambio si lo son el tiempo o/y el coste de desarrollo. Ello convierte a los lenguajes funcionales en candidatos especialmente idóneos para el desarrollo de prototipos tempranos o de herramientas laterales al desarrollo principal.

Crítica 3: dificultad del modelo

La crítica final consiste en afirmar que el modo funcional de programar es poco intuitivo para el programador y que sólo personas con una mente complicada pueden llegar a ser competentes programadores en ese paradigma.

Repitiendo la respuesta que se da en [Cun94] a esta cuestión, la misma crítica podría hacerse de la programación en general: los programadores somos, para el resto de los mortales, personas de mentes complicadas. Más bien habría que pensar que nuestro conocimiento de los computadores se ha producido dentro del marco imperativo y que hemos empleado muchos esfuerzos y muchos años en estudiar ese paradigma. Todo lo que suponga un modo distinto de pensar ha de resultar por fuerza antinatural. Seguramente, con una educación y entrenamientos adecuados, la programación funcional puede llegar a ser tan intuitiva como la que hoy conocemos mejor.

4 Estado actual de la programación funcional

La programación funcional ha alcanzado su madurez como paradigma de programación. Lenguajes como Standard ML, Miranda y Haskell cumplen todos los requisitos necesarios para la programación industrial, a saber: reúnen suficientes facilidades para la construcción de grandes programas y existen compiladores de calidad para las plataformas más habituales.

La programación funcional se enseña actualmente en muchas universidades, incluidas la mayoría de las españolas. Hay disponible una buena cantidad de textos didácticos para su aprendizaje [BW88, Rea89, Wik87, FH88, Hol91] y cada año se edita alguno nuevo. También existen numerosos artículos introductorios que dan cuenta de los aspectos más importantes y de la historia y evolución del paradigma (ver p.e. [Hud89, Hug90]).

Para una primera toma de contacto, puede ser conveniente hacerse con el material de Gofer y experimentar con él. El lenguaje Gofer (GOod For Equational Reasoning) es un subconjunto de Haskell desarrollado por un estudiante de Oxford, actualmente en Nottingham, como parte de su tesis doctoral. Están disponibles como libre distribución el compilador y la documentación asociada [Jon92], y existen versiones del mismo para PC's, Macintosh'es y estaciones SUN (ftp anónimo a ftp.cs.nott.ac.uk, directorio nott-fp/languages/gofer).

El desarrollo de Haskell fue promovido con la finalidad de facilitar la difusión del paradigma funcional. El objetivo era tener un lenguaje standard que reuniera las facilidades estables del estado del arte, y

⁶Es interesante matizar que este "más", para la mayoría de los algoritmos, consiste en una mera constante multiplicativa independiente del tamaño del problema. Es decir, en términos de costes asintóticos, la magnitud $\mathcal{O}(f(n))$ sería la misma para ambos paradigmas.

para el cual se crearan compiladores de calidad. Otro requisito para facilitar la difusión era que ésta fuera libre. En el momento de escribir estas líneas, existen al menos tres compiladores que reúnen estas características: el desarrollado por la Universidad de Glasgow (ftp anónimo a ftp.dcs.gla.ac.uk: pub/haskell), el de la Universidad de Yale (haskell.cs.yale.edu: pub/haskell) y el de la Universidad de Chalmers (ftp.cs.chalmers.se: pub/haskell).

Sin perjuicio del grado de madurez descrito, la programación funcional —al igual que el otro paradigma declarativo mencionado al comienzo de este artículo, la programación lógica— sigue despertando el interés de los investigadores. Existen varias conferencias anuales o bianuales asentadas en el área, y también publicaciones internacionales específicas. En España hay numerosos grupos activos en ambos paradigmas y en su combinación en lenguajes híbridos lógico-funcionales, y cada año desde 1991 se celebra una Conferencia sobre Programación Declarativa (PRODE).

Gran parte de los esfuerzos se dirigen a mejorar las técnicas de compilación y al diseño de arquitecturas no convencionales que permitan ejecutar programas funcionales con mayor eficiencia. La explotación del paralelismo potencial del paradigma es en muchos casos la idea central (un ejemplo reciente es [KG93]).

Otra área distinta de la anterior es la incorporación, al nivel del lenguaje, de construcciones para expresar de forma explícita el paralelismo y el no determinismo. Los programas funcionales son, por su propia naturaleza, deterministas. Por esa razón son poco adecuados para diseñar sistemas reactivos que han de responder de forma no determinista a uno entre varios eventos externos posibles (p.e. un sistema operativo). Las extensiones propuestas van en el sentido de dotar a los lenguajes funcionales de estas capacidades [Hen82, Kel89].

La transformación de programas funcionales ha sido objeto de atención casi desde los comienzos. El objetivo aquí es poder construir algoritmos mediante refinamientos sucesivos. Se parte de una versión ingenua del mismo, cuya corrección es obvia por sí misma, y gradualmente se transforma, mediante transformaciones que preservan la corrección, a versiones más eficientes (ver p.e. [Bir89, BJM91]).

Los sistemas de tipos con polimorfismo y sobrecarga son todavía muy recientes y se continúan haciendo propuestas para su extensión en diversas direcciones [Jon93, CP94].

Finalmente, un concepto proveniente de la teoría de categorías, las *mónadas*, está encontrando numerosas aplicaciones en la programación funcional, especialmente en aquellas áreas que limitan con conceptos imperativos (estado, entrada/salida, excepciones), y está siendo objeto de gran atención en los últimos tiempos (para una introducción, ver [Wad92]).

References

- [Bac78] J. Backus. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs. *Comm. ACM*, 21(8):613–641, August 1978.
- [Bar84] J. G. P. Barnes. *Programming in Ada*. International Computing Press, 1984.
- [Bir89] R. S. Bird. Lectures on constructive functional programming. In *Constructive Methods in Computer Science*, NATO ASI, pages 150–217. Springer-Verlag. NATO ASI Series, vol. F55, 1989.
- [BJM91] J. B. Banâtre, S. B. Jones, and D. Le Métayer. *Prospects for Functional Programming in Software Engineering*. Springer-Verlag, 1991. Capítulo 5.
- [BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.
- [Car89] L. Cardelli. Typeful Programming. DEC SRC Report no. 45, 1989.
- [CP94] S. Cléricali and R. Peña. Generic Classes Parameterized by Data Structures. In *GULP-PRODE'94: Joint Conference on Declarative Programming. Vol. 2*, pages 423–437, 1994.
- [Cun94] H. C. Cunningham. Notes on functional programming with gofer. Tech. Report UMCIS-1994-01, University of Mississippi, 1994.
- [FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988. Capítulo 17.

- [GNPP94] L. A. Galán, M. Núñez, C. Pareja, and R. Peña. Homomorphisms, non Homomorphisms and Transformations Between them. In *GULP-PRODE'94: Joint Conference on Declarative Programming. Vol. 2*, pages 393–407, 1994.
- [Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and Its Applications*, pages 177–192. Cambridge University Press, 1982.
- [HF92] P. Hudak and J. H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.
- [HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ml. Technical Report Report ECS-LFCS-86-2, Edinburgh University, 1986.
- [Hol91] I. Holyer. *Functional Programming with Miranda*. Pitman, 1991.
- [HPJ94] P. Hudak and M. Peyton-Jones. Haskell vs. ada vs. c++ vs. awk vs. ... An Experiment in Software Prototyping Productivity. ftp anónimo a haskell.cs.yale.edu, Jul. 1994.
- [HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell. version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
- [Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, Sept. 1989.
- [Hug90] J. Hughes. *Why Functional Programming Matters*, pages 17–43. Research Topics in Functional Programming. Addison-Wesley, 1990. (Ed.) D. A. Turner.
- [Jon92] M. P. Jones. *GOFER Manual*. Oxford University Computing Laboratory, 1992.
- [Jon93] M. P. Jones. *GOFER 2.28 release notes*. Oxford University Computing Laboratory, 1993.
- [Kel89] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Pitman, 1989.
- [KG93] H. Kuchen and K. Gladitz. Parallel Implementation of Bags. In *Conference on Functional Prog. Lang. and Comp. Arch. FPCA'93*. ACM Press, 1993.
- [Mil84] R. Milner. A Proposal for Standard ML. In *Proceedings of the ACM Conf. on Lisp and Functional Programming*, pages 184–197, 1984.
- [oD81] U.S. Department of Defense. *The Programming Language Ada, Reference Manual*. LNCS 106. Springer Verlag, 1981.
- [Rea89] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
- [Tur85] D. A. Turner. *Miranda: A Non-Strict Functional Language with Polymorphic Types*, pages 1–16. LNCS 201. Springer-Verlag, Berlin, 1985.
- [Tur86] D. A. Turner. An Overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, Dec. 1986.
- [Wad92] P. Wadler. The Essence of Functional Programming. *19th Symp. on Principles of Programming Languages*, January 1992.
- [Wik87] A. Wikström. *Functional Programming Using Standard ML*. Prentice-Hall, 1987.