75.31 Teoría de Lenguajes

Haskell
λ

Albani Francisco 84891

2.^{do} cuatrimestre 2006

Resumen

El presente informe comienza presentando al lector algunos conceptos previos necesarios para adentrarse en el mundo de *Haskell*. Se dan breves ideas sobre la importancia del *Paradigma de Programación Funcional* y sobre el significado del *Cálculo Lambda*. Luego, aportándole un toque de color al informe, se recorre el contexto histórico del nacimiento de *Haskell*. Finalizada la introducción, se presentan las características más importantes del lenguaje de una forma principalmente teórica, para luego dejarle lugar a un enfoque más técnico, donde se repasan varios de los elementos que constituyen a *Haskell*, aportando ejemplos allí donde vinieran a ilustrar y dar soporte a las ideas. Es quizá necesario advertir al lector que en ningún momento el presente informe intenta servir de tutorial ni profundizar demasiado en todos los detalles.

Índice

Parte I
Introducción

1. ¿Qué es *Haskell*?

Haskell is probably doomed to succeed.
Tony Hoare[1]
Lisp was the most beautiful language in the world
At least up until Haskell came along.
Larry Wall[2]

Haskell es un lenguaje de programación de amplio espectro inspirado fuertemente en el *Cálculo Lambda*, que responde al *Paradigma de Programación Funcional*. Su semántica no estricta, su pureza funcional y el manejo de tipos que posee, lo destacan especialmente del resto de los lenguajes.

Su diseño es llevado a cabo por un comité formado en **1987** para el cual uno de los objetivos más importantes siempre ha sido la **elegancia matemática**.

Programar grandes sistemas de software es caro y dificultoso. El mantenimiento de esos sistemas lo es aún más. Los lenguajes de programación funcionales, como Haskell, pueden hacerlo más fácil y barato. Es común escuchar decir, entre quienes han probado Haskell, que el sistema de tipos es muy bueno a la hora de evitar varias clases errores bastante comunes en la práctica. Inclusive entre quienes no pueden utilizar Haskell en sus proyectos, el haberlo aprendido les a ayudado a pensar los problemas desde otra perspectiva, y construir mejores soluciones en otros lenguajes.

Según sus creadores, Haskell ofrece:

- Incremento en la productividad del programador.
- Código más corto, claro y fácil de mantener.
- Menor cantidad de errores y mayor confiabilidad.
- Una *distancia semántica* menor entre el programador y el lenguaje.
- Eliminación de una altísima cantidad de interacciones imprevistas, producto del riguroso control de efectos colaterales.

[1] Científico británico famoso por, entre otras cosas, la invención del *Quicksort*, el algoritmo de ordenamiento más utilizado en el mundo.

[2] Programador californiano famoso por ser el autor del lenguaje de programación PERL y por haber recibido el primer premio para el *Avance del Software Libre* de la *Fundación para el Software Libre*.

2. Paradigma de Programación Funcional

Haskell, Lisp, and SQL are the only programming languages that I've seen where one spends more time thinking than typing.
Philip Greenspun[3]

2.1. Contexto histórico

En la década de **1930**, Alonzo Church y Stephen Kleene inventan el *Cálculo Lambda*, un marco teórico que describe las funciones y su evaluación. Aunque es una abstracción matemática y no un lenguaje de programación, ha formado la base de casi todos los lenguajes funcionales. La *Lógica Combinacional*, creada por Moses Schönfinkel y luego desarrollada por Haskell Curry con objetivos más generales, precedió al Cálculo Lambda en su invención pero comenzó a tomar importancia a partir de la década de **1960** cuando la Ciencia teórica de la Computación comenzó a interesarse por ella.

En **1978**, John Backus[4], al recibir el *Premio Turing*[5], tituló su conferencia: *"¿Puede la programación ser liberada del estilo de* ***von Neumann***[6] *? Un estilo Funcional y su álgebra de programas"*[Bac78], posicionando a la programación funcional como un ataque radical hacia todo lo establecido en el mundo de la programación. Semejante proposición pronunciada por un gigante en el campo, situó en el mapa a la programación funcional de una nueva manera: como una herramienta práctica en vez de una mera curiosidad matemática.

En **1984**, John Hughes, publicó un artículo titulado *¿Por qué la programación funcional importa?*[Hug89], donde pretendía demostrarle al "mundo real"que la programación funcional era de vital importancia, y también ayudar a los programadores funcionales a explotar sus ventajas al máximo, especificando claramente cuáles eran esas ventajas.

He aquí la introducción de aquel artículo:

> A medida que el software se vuelve más y más complejo, es más y más importante estructurarlo bien. El software bien estructurado es fácil de escribir, fácil de depurar, y provee una colección de módulos que pueden ser reutilizados para reducir futuros costos de programación. Los lenguajes convencionales imponen límites conceptuales

[3] Profesor de *Ingeniería de Software para aplicaciones de Internet* e *Ingeniería Eléctrica* en el Instituto de Tecnología de Massachusetts (M.I.T.).

[4] Backus lideró el equipo que desarrolló FORTRAN, entre otras cosas muy trascendentes.

[5] El *Premio Turing* es considerado por muchos como el Premio Nobel de la Computación. Es otorgado anualmente por la *Association for Computing Machinery* a quienes hayan contribuido de manera trascendental al campo de las ciencias computacionales.

[6] La mayoría de los lenguajes de programación entran en esta categoría. Ejemplos son FORTRAN, C y Java.

> en la forma en la que los problemas pueden modularse. Los lenguajes funcionales empujan esos límites hacia atrás. [...]
>
> Siendo la modularidad la clave para una programación exitosa, los lenguajes funcionales son de vital importancia para el "mundo real".

2.2. Características

La programación funcional recibe ese nombre porque un programa se construye solamente a partir de funciones. El propio programa principal es escrito como una función que recibe una entrada como argumento y entrega un resultado como salida. Típicamente, la función principal es definida en términos de otras funciones, que a su vez son definidas en los términos de otras funciones, hasta llegar al último nivel, donde las funciones son primitivas del lenguaje. Todas estas son funciones en el más estricto sentido matemático; lo que inclusive se ve reflejado en el momento de su definición, que se lleva a cabo por medio de ecuaciones.

La programación funcional está contenida en un paradigma más general llamado *Programación Declarativa*, cuya principal característica es la de describir los objetivos de un algoritmo sin especificar su implementación[7]. De ahí es de donde los programas funcionales heredan la cualidad de no contener declaraciones de asignación; una variable nunca cambia su valor inicial. Más generalmente, los programas funcionales no contienen ningún tipo de efecto colateral. Una llamada a una función no puede tener otro efecto más que el cómputo de su resultado. Esto elimina una de las principales fuentes de errores, y también hace que el orden de ejecución sea irrelevante, ya que si ningún efecto colateral puede alterar el valor de una expresión, entonces puede ser evaluada en cualquier momento. Esto releva al programador de la carga de tener que controlar el flujo.

Debido a que las expresiones pueden ser evaluadas en cualquier momento, las variables pueden reemplazarse libremente por sus valores y viceversa, sin alterar el resultado del programa. Esta característica es conocida como *Transparencia Referencial*, y hace a los programas funcionales matemáticamente más tratables.

2.3. Modularidad

El diseño modular trae consigo una gran mejora en la productividad. Primero, pequeños módulos pueden ser escritos fácil y rápidamente. Segundo, los módulos de propósito general pueden ser reutilizados, llevando a un desarrollo más rápido de aplicaciones subsecuentes. Tercero, los módulos de un programa pueden ser probados independientemente, ayudando a reducir el tiempo de depuración.

Sin embargo, hay algo importante que usualmente es olvidado. Cuando se escribe un programa modular para resolver un problema, primero se divide al problema en partes, se las resuelve y luego se combinan las soluciones. Las formas en las que el problema original puede ser dividido depende directamente

[7] No intenta ser una definición rigurosa; cada lenguaje refleja esta característica en mayor o menor medida.

de las formas en las que las soluciones pueden ser combinadas. Por lo tanto, para incrementar la capacidad de modular un problema, un lenguaje tiene que proveer nuevas formas de combinar soluciones.

La programación funcional provee dos nuevas formas de combinar soluciones: las funciones de alto orden, y la semántica no estricta. La clave del poder de la programación funcional es permitir mejoras significantes en la modularidad.

3. Cálculo Lambda λ

El Cálculo Lambda es un sistema formal de notación para investigar y describir la definición y aplicación de funciones matemáticas e inclusive de algoritmos. Fue diseñado por Alonzo Church y Stephen Kleene en la década del '30. Church lo utilizó para, en 1936, dar una respuesta negativa al famoso problema de decisión conocido como *Entscheidungsproblem*[8]. Cualquier función computable puede ser expresada y evaluada con este formalismo. Es, por lo tanto, equivalente a una *Máquina de Turing*.

Lo que matemáticamente se expresaría $f(x) = x^2$, en el Cálculo Lambda se expresaría $\lambda x.x^2$. La especialización de $f(x)$ en un valor a arbitrario, $f(a)$, se expresa $(\lambda x.x^2)\, a$; resultando en la equivalencia entre a^2 y $\lambda x.a^2$.

Es importante destacar que en el Cálculo Lambda (y en Haskell) no hay una distinción entre valores y funciones, lo que permite utilizar estas últimas como argumentos de otras funciones. Este concepto se ampliará en la Sección 7.

El conjunto de todas las *expresiones lambda* puede ser descripto por la siguiente gramática libre de contexto:

$$e ::= x \mid \lambda x.e \mid e_o\, e_1$$

Donde:

- x es un identificador que representa a una variable en el sentido matemático.
- e, e_0 y e_1 son expresiones lambda.

[8] Consiste en averiguar si existe un algoritmo que decida si una declaración lógica de primer orden, es universalmente válida o no.

4. Un poco de Historia

4.1. El nacimiento

En Septiembre de **1987**, una reunión impulsada por Simon Peyton Jones, Paul Hudak y Philip Wadler, se llevó a cabo en la *Conferencia de Lenguajes de Programación Funcional y Arquitecturas de Computadores* (FPCA)[9] en Portland, Oregon, con el objetivo de discutir la desafortunada situación que atravesaba la comunidad de lenguajes de programación funcional: existía más de una docena de lenguajes puros y no estrictos, todos muy similares en poder expresivo y semántico, y el uso más extenso de esta clase de lenguajes estaba siendo obstaculizado por la carencia de un lenguaje común. Se decidió que un comité debía formarse para diseñar tal lenguaje, permitiendo una mejor comunicación de ideas, una base estable para el desarrollo de aplicaciones reales, y un vehículo a través del cual otros pudieran sentirse motivados a usar lenguajes funcionales.

En un principio, la idea era tomar como base un lenguaje ya existente, para luego evolucionar en la dirección más conveniente. De todos los lenguajes no estrictos en desarrollo, el más maduro, por lejos, era Miranda: un lenguaje puro, bien diseñado, con muchas cosas en común a las ideas del comité y una implementación robusta producida por la empresa de David Turner, Research Software Ltd.

Luego de una breve y cordial charla, Turner no aceptó que Miranda sea adoptado como punto de partida para la creación de este nuevo lenguaje. Sus objetivos eran distintos a los del comité: estaba fuertemente determinado a mantener un solo standard y no quería que existieran múltiples dialectos de Miranda en circulación. Inclusive pidió que el nuevo lenguaje fuera lo suficientemente distinto de tal forma que no fueran confundidos. Tampoco aceptó la invitación a formar parte del comité.

Este impedimento, que obligó a comenzar el desarrollo desde cero, permitió contemplar enfoques más radicales para muchos aspectos del diseño. Muchas características actuales de Haskell hubieran sido improbables de haber sido desarrollado a partir de Miranda. Sin embargo, Haskell tiene una considerable deuda con Miranda, tanto por la inspiración general como por algunos elementos específicos adoptados libremente para satisfacer algunas características del diseño emergente.

La primer reunión física llevada a cabo luego de la FPCA, fue en Enero del '88, en la Universidad de Yale, ubicada en New Heaven, Connecticut. Lo primero fue establecer los siguientes objetivos para el lenguaje:

- Debía ser adecuado para la enseñanza, la investigación y el desarrollo de grandes aplicaciones.
- Debía ser completamente descripto a través de la publicación de una sintaxis formal y una semántica.

[9] Functional Programming and Computer Architecture Conference.

- Debía ser disponible libremente. Debía permitirse que cualquier persona pudiera implementar el lenguaje y distribuirlo a quien quisiera.
- Debía poder ser usado como base para la investigación de lenguajes.
- Debía estar basado en ideas que gozaran de un consenso considerable.
- Debía reducir la innecesaria diversidad entre los lenguajes funcionales.

Los últimos dos objetivos reflejaban la intención de que el lenguaje fuera un tanto conservador, en lugar de pretender grandes innovaciones; si bien luego todo resultó bastante diferente.

4.2. El Bautismo

La elección del nombre se llevó a cabo de la siguiente manera: cada miembro del comité podía proponer todos los que quisiera, para luego ser escritos en un pizarrón. Al final de este proceso, los siguientes nombres aparecieron:

- Semla
- Haskell
- Vivaldi
- Mozart
- CFL (Common Functional Language)
- Funl 88
- Semlor
- Candle (Common Applicative Notation for Denoting Lambda Expressions)
- Fun
- David
- Nice
- Light
- ML Nouveau (o Miranda Nouveau, o LML Nouveau, o ...)
- Mirabelle
- Concord
- LL
- Slim
- Meet
- Leval
- Curry
- Frege
- Peano
- Ease
- Portland
- Haskell B Curry

Luego de considerables discusiones a cerca de cada uno de los nombres, cada persona fue libre de tachar uno que no le gustara. Cuando esto fue terminado, solamente quedó uno.

Ese nombre fue *Curry*, en honor al lógico matemático Haskell Brooks Curry. Pero al notar que el nombre se prestaba para varios juegos de palabras burlescos, y estaba relacionado con el actor *Tim Curry*, muy famoso en aquel momento por su papel de científico loco en *The Rocky Horror Picture Show*, se optó por establecer *Haskell* como el nombre definitivo.

Tiempo después, Hudak visitó a la viuda de Haskell Curry, para preguntarle si tenía problemas en que se use el nombre de su marido, a lo que respondió negativamente. Luego de eso, la Señora Curry asistió a una charla de Hudak, sobre Haskell (el lenguaje); y más allá de no haber entendido una sola palabra de lo que se habló, se mostró muy agradecida. Lo más memorable de lo que dijo fue: *"¿Saben?, a Haskell nunca le gustó el nombre Haskell"*.

Parte II
Características Especiales

5. Evaluación perezosa

Efficiency is intelligent laziness.
David Dunham

Dentro del conjunto de las *Estrategias de Evaluación*, se encuentran principalmente dos grandes subconjuntos: las estrategias de *Evaluación Estricta* y las estrategias de *Evaluación no-Estricta*. En el primer conjunto, encontramos a las estrategias donde las expresiones son evaluadas en el momento en el que se las encuentra[10]. En el segundo conjunto, tenemos a las estrategias donde la evaluación de una expresión se difiere hasta un momento posterior. Un ejemplo de estas es *call-by-name*[11], que consiste en reemplazar en el momento en el que se necesite, en el cuerpo del contexto, las demandas a la expresión por la expresión directamente. Una versión mejorada de *call-by-name* es *call-by-need*, más conocida como *evaluación perezosa* o *laziness*, que solo evalúa una sola vez la expresión para evitar repetir los cálculos.

En Haskell, de forma predeterminada y con algunas excepciones, las expresiones se evalúan por *call-by-need*. El lenguaje provee los mecanismos necesarios para especificar la estrategia según sea necesario.

La Evaluación perezosa tiene sus costos. Usualmente es menos eficiente que las estrictas, debido a la contabilidad adicional requerida para satisfacer todas sus características. El costo es significante pero al haberse considerado un factor constante en la mayoría de los casos, esta opción prevaleció por encima del resto. Sucede también que es muy difícil, inclusive para los programadores expertos, predecir el comportamiento espacial de un programa perezoso especialmente cuando este factor no es constante, lo que condujo a replantear el diseño de tal forma de permitir ciertos comportamientos estrictos.

Por otro lado, varios lenguajes estrictos han comenzado a incluir características perezosas al reconocer las virtudes que tiene en muchos casos. Como un resultado de esto, la división entre lo estricto y lo perezoso ha dejado de verse como una cuestión de *blancos y negros*.

[10] Estas son las elegidas por la mayoría de los lenguajes. Vale la pena aclarar que aún en estos, en general, las estructuras if-then-else solo evalúan la expresión que corresponde al camino que el flujo a de tomar.

[11] Esta estrategia es más un concepto teórico que un recurso práctico.

6. Pureza funcional

Once we were committed to a lazy language, a pure one was inescapable.
A history of Haskell [His07].

Una consecuencia inmediata de la evaluación perezosa es que el orden en el que son evaluadas las expresiones, no está dado por el orden de sus apariciones, como se esperaría en un lenguaje estructurado, si no que está dado por el orden en el que van siendo requeridas. Esto imposibilita realizar de forma confiable operaciones de entrada/salida o que impliquen algún otro tipo de efecto colateral a la llamada de alguna función. Una vez elegida la evaluación perezosa, el comité no pudo escapar a construir Haskell en base a funciones puras. Esto es, en otras palabras, limitarlas a tomar parámetros y devolver resultados sin poder provocar ningún tipo de efecto colateral a su ejecución en el más estricto sentido matemático.

Otra característica importante que hace a Haskell un lenguaje puro, es excluir la posibilidad de realizar cambios destructivos sobre los datos. Esta es quizá una de las diferencias más radicales con respecto a la programación estructurada, donde cambiar el valor de una variable es uno de los recursos más comunes a la hora de escribir algoritmos. Una consecuencia de esto es que las variables refieren siempre a valores inmutables y persistentes. Se conoce como *Transparencia referencial* a la propiedad de un lenguaje en el cual todas sus expresiones pueden ser reemplazadas con su resultado sin cambiar el comportamiento del programa resultante. Esto está en estrecha relación con la pureza de las funciones, pues ante mismos parámetros, devuelven siempre el mismo resultado, permitiendo implementar una técnica de optimización conocida como *Memoization*, que consiste en recordar los resultados de las llamadas anteriores para no tener que volver a computarlos. A los lenguajes donde, por el contrario, no es posible realizar el reemplazo de sus expresiones por sus resultados sin alterar el resultado, se les atribuye la propiedad de *Opacidad referencial*.

La cara oscura de esta característica es que estaríamos limitados a programar sistemas que respondan a un modelo muy simple de entrada, procesamiento y salida de datos, donde la única interfaz con el exterior serían entrada y salida. En un principio, el sistema de entrada y salida de Haskell era muy torpe y considerado una fuente de vergüenza para sus diseñadores. Pero finalmente resultó ser la madre de una invención que es considerada como la mayor contribución de Haskell al mundo de los lenguajes de programación: *entrada/salida monádica*[12].

[12] La sección 8 está dedicada a este concepto.

7. Programación de Alto Orden

Haskell hereda el rasgo del *Cálculo Lambda* de no diferenciar sustancialmente a las funciones de los valores. Se suele decir que las funciones, son *valores de primer orden*; esto significa que están al mismo nivel que los *enteros*, los *caracteres* y cualquier otro tipo de dato. Pueden ser ligadas a variables, almacenadas en colecciones, ser pasadas como parámetro de otras funciones y ser devueltas como valor de retorno de una función. Los ejemplos más comunes en los lenguajes funcionales de funciones de *Alto Orden* son `map`, `filter` y `foldl` o `foldr`.

Las funciones de *Alto Orden* pueden ser vistas como una forma de ejecución diferida, donde una función es definida en un contexto y pasada a otro diferente para luego ser invocada. La diferencia con las funciones comunes encontradas en otros lenguajes, es que las funciones de *Alto Orden* representan abstracciones lambda anónimas, permitiendo que el contexto invocador no necesite conocer el nombre del la función.

En otros lenguajes existen las llamadas *Clausuras lexicográficas*, que complementan la idea anterior anexando a la función todo el entorno de su definición, permitiéndole acceder a éste inclusive cuando es invocada desde un contexto distinto.

8. Mónadas

Haskell is the world's finest imperative programming language
Simon Peyton Jones.
[Pey00]

Las características anteriores hacen a Haskell un lenguaje muy elegante. Pero todas ellas eluden hablar de cosas como operaciones de *Entrada/Salida*, detección y recuperación de errores, concurrencia e interacción con librerías o componentes escritos en otros lenguajes. Todas ellas un tanto traumáticas y a la vez cruciales para que una aplicación sea de calidad y verdaderamente útil.

La familia de lenguajes funcionales que responden a la estrategia de evaluación conocida como *call-by-value*, en general toman un enfoque pragmático, adoptando algunos rasgos de los lenguajes imperativos. Simplemente tienen funciones que realizan muchas de estas tareas provocando efectos colaterales a su ejecución. Claro que dejan de ser funciones en el sentido matemático, pero en la práctica este enfoque ha probado funcionar, si también se tiene en cuenta la especificación del orden de evaluación como parte del diseño. Esta es la forma en la que la mayoría de los lenguajes (funcionales o no) encaran los problemas que implican interacción con el mundo exterior.

Los lenguajes, como Haskell, que utilizan la estrategia de evaluación *call-by-need*, no pueden adoptar el enfoque anterior porque el orden de evaluación está deliberadamente no-especificado. Una forma de mostrar este impedimento es considerar el siguiente ejemplo:

```
xs = [printChar 'a', printChar 'b']
```

En un lenguaje *call-by-value*, evaluar esa ligadura tendría dos efectos colaterales: imprimir en pantalla la letra *a* y la letra *b*. Pero en Haskell, las llamadas a `printChar` solo se ejecutarían si los elementos de la lista son evaluados. Por ejemplo, si el único uso de `xs` es en una llamada a `length xs`, entonces nada sucedería en la pantalla, porque `length` no necesita evaluar los elementos de una lista para conocer su longitud.

La conclusión es que la evaluación perezosa y los efectos colaterales, desde un punto de vista práctico, son incompatibles.

Durante mucho tiempo esta situación fue embarazosa para la comunidad *lazy*, hasta que una solución sorpresiva apareció: **las Mónadas**, un concepto derivado de la Teoría de Categorías, una de las ramas más abstractas de la matemática, que ha resultado ser la contribución más importante que Haskell ha hecho al mundo de los lenguajes de programación.

Las Mónadas, permiten estructurar operaciones que deben ser realizadas en un determinado orden, siendo la *Entrada/Salida* el campo dónde más útiles han resultado ser. La idea consiste en que una función, si bien no puede directamente provocar efectos colaterales, puede construir un valor que describa el efecto colateral deseado que quien la llama debería provocar.

Se puede pensar a una mónada como una tupla de 3 elementos $(M, return, bind)$ donde M es un constructor de tipo[13] y, tanto $return$ como $bind$, son funciones polimórficas cuya definición puede expresarse así:

$$return : a \to M\, a$$

$$bind : M\, a \to (a \to M\, b) \to M\, b$$

Para que una instancia particular de lo anterior resulte consistente con el marco teórico que le da vida, las definiciones de $return$ y $bind$ deben satisfacer las siguiente 3 leyes:
(se adopta la notación >>= para significar la versión infija de bind)

- $return$ debe preservar toda la información de su argumento:

$$(return\, x) >>= f \equiv f\, x$$

$$m >>= return \equiv m$$

- $bind$ debe ser asociativa:

$$(m >>= f) >>= g \equiv m >>= (\lambda x.\, (f\, x >>= g))$$

Donde:

- f, $g : a \to M\, a$
- m es una mónada de la forma $M\, a$.
- la notación $\lambda x.\, e$ corresponde a una función anónima que convierte valores de x a la expresión e. La Sección 10.7 está dedicada a este tipo de expresiones.

En palabras, el significado y objetivo de $bind$ es secuenciar dos acciones donde la segunda necesita conocer el resultado de la primera. Esa *información* es de tipo a y viaja contenida en la mónada. Por otro lado, el significado y objetivo de $return$ es servir de acción final en una secuencia de acciones[14].

En la sección 15 se verá de qué forma todo esto encaja en el sistema de tipos de Haskell.

[13] Puede considerarse a M como un *contenedor* también.
[14] No es casualidad que su nombre signifique *retornar*.

Parte III
Características Técnicas

9. Sistema de Tipos

Haskell utiliza un sistema de chequeo estático de tipos. Esto significa que a cada expresión se le asigna un tipo en el momento de compilación. En el caso en el que una función, que espera un argumento de un cierto tipo, recibe uno de tipo incorrecto, se producirá un error en tiempo de compilación impidiendo la construcción del programa. Es importante destacar que la gestión de memoria es realizada por un *colector de basura* y en ningún momento el programador se ve involucrado en la reserva o liberación explícita de la misma.

Haskell provee un conjunto de tipos básicos donde se pueden encontrar los comunes a la mayoría de los lenguajes: `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`, `Real` y `Floating`.

Forman parte también del conjunto básico las *n-Tuplas*, arreglos de una cantidad fija de elementos heterogéneos: los pares `(a, b)`, las ternas `(a, b, c)`, etc.

Las listas son quizá el tipo de dato más utilizado en los lenguajes funcionales y es por eso que, más adelante, se les dedicarán las secciones 13.4 y 15.1.2. Lo único que se dirá ahora es lo necesario para comprender algunos de los ejemplos a continuación: *La notación de una lista cuyos elementos son de tipo* `a` *, es* `[a]`.

9.1. Sinonimia de Tipos

Haskell provee un mecanismo para establecer tipos sinónimos. Resulta útil para renombrar tipos de uso muy común que cobran un significado propio. A continuación, algunos ejemplos:

```
type Nombre = String
type Edad = Int
type Persona = (Nombre, Edad)
type Agenda = [Persona]

type Punto2D = (Floating, Floating)
type Punto3D = (Floating, Floating, Floating)
type Trayectoria2D = [Punto2D]
```

Debe quedar claro que un sinónimo **no** es un nuevo tipo. Todas aquellas funciones que esperen recibir un argumento de tipo `Agenda`, funcionarán también con argumentos del tipo `[(Nombre, Edad)]`.

10. Funciones

Siendo Haskell un lenguaje funcional, es natural que las funciones sean el motor de todo. Un programa es una gran función cuyos cálculos se definen mediante otras funciones, cuyas definiciones también dependen de otras funciones y así hasta llegar a las primitivas del lenguaje. Siendo tan importantes, es esperable que se provean múltiples mecanismos simples para definirlas y manipularlas.

Como ya se comentó en la Sección 7, en Haskell las funciones son valores de primer orden; están al mismo nivel que los números, los caracteres o cualquier otro tipo de dato.

Una función se define mediante ecuaciones. Las ecuaciones se componen de dos partes, principalmente. La primera, es donde se especifica el nombre de la función y sus argumentos; la segunda, contiene el cuerpo ejecutable. Un simple ejemplo podría ser el siguiente:

```
duplicar x = x * 2
```

En el siguiente caso, se muestra una función que se define mediante múltiples ecuaciones:

```
opcion 1 = "Iniciar"
opcion 2 = "Cargar"
opcion 3 = "Grabar"
opcion 0 = "Salir"
opcion n = "Opción incorrecta"
```

10.1. *Pattern Matching*

Es necesario a partir de ahora comprender que en Haskell todos los argumentos se expresan como patrones. Al ser llamada la función, el compilador comienza con la primer ecuación hasta la última a probar si los argumentos con los que ha sido invocada coinciden con alguno de los patrones. Una vez encontrada la ecuación, el compilador procede a ligar los identificadores libres del patrón. En el primer ejemplo de la Sección 10, cualquier parámetro coincide con el patrón y es ligado a `x`. En el segundo, una llamada a `opcion 6`, solo coincidiría con la última ecuación, y `n` pasaría a valer 6. Por el contrario, una llamada a `opcion 2`, coincidiría con la segunda ecuación y no se seguiría probando con el resto.

Si, por ejemplo, se tiene una función que recibe una terna y se quiere devolver alguno de sus elementos, se puede usar *Pattern Matching* para descomponerla en sus componentes:

```
posicionX (x, y, z) = x
```

Es importante hacer notar que si bien esta función parecería tener tres argumentos, guiándose por la sintaxis de otros lenguajes, es en realidad uno solo pero expresado por un patrón.

Los parámetros formales son también patrones, con la particularidad que nunca fallan. Siempre es posible ligar el parámetro `x` a cualquier expresión.

Los patrones que nunca fallan, como los parámetros formales, se dice que son *irrefutables*. En el mismo sentido, los que sí pueden fallar se conocen como *refutables*. Hay otros dos tipos de patrones *irrefutables* importantes: *As-patterns* y *Wild-cards*.

10.1.1. *As-patterns*

Es a veces conveniente darle un nombre al patrón para referirse a él en la parte derecha de una ecuación. Por ejemplo, en una función de la forma:

```
f (x, y) = if x == 0 then (y, y) else (x, y)
```

se puede ver que `(x, y)` aparece una vez como patrón (a la izquierda de la ecuación), y otra vez como expresión (a la derecha). Para mejorar la legibilidad (o para otros fines) es preferible usar la siguiente técnica:

```
f s@(x, y) = if x == 0 then (y, y) else s
```

donde se a `s` se la liga a todo el patrón mediante el símbolo `@`.

Estos patrones siempre coinciden, más allá de que los subpatrones puedan fallar.

10.1.2. *Wild-cards*

Otra situación común es cuando se desea construir un patrón con partes irrelevantes. Por ejemplo, una función que reduce una terna a un par, no necesita conocer al último elemento:

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

Puede interpretarse al símbolo `_` como un comodín que puede coincidir con cualquier forma, pero no establece ningún tipo de ligadura. Es por eso que pueden usarse múltiples veces en un mismo patrón sin provocar ambigüedades, cosa que no es posible con los parámetros formales.

10.1.3. Construcciones *case of*

En Haskell no se cumple la premisa que dice que todo lo que se puede hacer con un **case**, se puede hacer con un **if**. Sí se cumple la inversa. La gran diferencia es que en los primeros pueden utilizarse patrones y en los segundos solo expresiones booleanas.

El *Pattern matching* provee un mecanismo de consignar el control basándose en las propiedades estructurales de un valor. Cada una de las ecuaciones que definen una función presentan un patrón distinto y significan un camino a seguir distinto. No siempre es deseable que cada nuevo patrón posible introduzca una nueva ecuación, es por eso que Haskell posee una forma de expresar esto de una forma más concisa, las construcciones *case of*.

Una definición de la forma:

```
f pat11 ... pat1k = e1
...
f patn1 ... patnk = en
```

donde cada patij es un patrón, es semánticamente equivalente a:

```
f x1 x2 ... xk = case (x1, ..., xk) of
                        (pat11, ..., pat1k) -> e1
                        ...
                        (patn1, ..., patnk) -> en
```

10.2. Variables locales

Es a veces necesario simplificar el cuerpo de una función definiendo variables auxiliares que representan expresiones repetidas. Haskell provee dos formas de hacer esto: los espacios `let` y `where`. A continuación se muestra el clásico ejemplo de las raíces de un polinomio de segundo grado:

```
raices a b c =
   let det   = sqrt (b*b - 4*a*c)
       dos_a = 2*a
   in
       ((-b + det) / dos_a, (-b - det) / dos_a)
```

A veces es más cómodo dejar las variables para el final:

```
raices a b c = ((-b + det) / dos_a, (-b - det) / dos_a)
 where det   = sqrt (b*b - 4*a*c)
       dos_a = 2*a
```

10.3. Guardas

Una alternativa a la hora de definir funciones partidas, es hacerlo mediante guardas. Esta forma es mucho más parecida a la forma en la que se definen funciones partidas en la matemática. Por ejemplo, la sucesión de Fibonacci, cuya definición es la siguiente:

$$f(n) = \begin{cases} 0 & \text{para } n = 0 \\ 1 & \text{para } n = 1 \\ f(n-2) + f(n-1) & \text{para } n > 1 \end{cases}$$

Puede traducirse a Haskell de la siguiente manera:

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

Cada guarda es una condición booleana. Puede suceder que el compilador encuentre la ecuación cuyo patrón coincide con el parámetro y al entrar a la definición, los guardas no le permitan ejecutar ninguna sentencia. En este caso el compilador descartará la ecuación y continuará buscando. Dependiendo de cómo se ordenen, puede emitir una advertencia de solapamiento de patrones.

10.4. Tipo de las funciones

Uno de los lugares donde Haskell justifica ser matemáticamente elegante es en la notación de los tipos de las funciones. Así como en las matemáticas el tipo de una función se expresa $f : A \to B$, en donde A y B son conjuntos de valores (tipos), en Haskell se expresa `f :: a -> b`, dónde `a` y `b` representan tipos genéricos.

He aquí algunos ejemplos:

- `esMayuscula? :: Char -> Bool`
- `esPar? :: Integer -> Bool`
- `sumaElementos :: [Integer] -> Integer`
- `losDosPrimeros :: [Integer] -> (Integer, Integer)`

Cuando la función recibe más de un parámetro, la notación se extiende de la siguiente manera:

- `letraEsta? :: Char -> String -> Bool`
- `sonElMismo? :: Integer -> Integer -> Bool`
- `uneListas? :: [Integer] -> [Integer] -> [Integer]`

10.5. Funciones polimórficas

En los ejemplos anteriores solo se vieron definiciones sobre tipos específicos. Es muy común que las funciones operen sobre más de un tipo o sobre estructuras que contienen tipos irrelevantes para la operación en cuestión. Estas funciones se conocen como polimórficas, pues su tipo es válido para muchas formas. Los clásicos ejemplos son las operaciones sobre listas. A continuación se muestra el tipo de varias funciones que operan sobre listas:

- `length :: [a] -> Int`
- `map :: (a -> b) -> [a] -> [b]`
- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `filter :: (a -> Bool) -> [a] -> [a]`

Donde `a` y `b` representan tipos genéricos.

Otro ejemplo más interesante, es el de una función que devuelve la composición de dos funciones que recibe como parámetro:

```
componer :: (b -> c) -> (a -> b) -> a -> c
```

10.6. *Currying*

Currying es la técnica de transformar una función cuyos argumentos están contenidos en un patrón de la forma `(x, y, ..., z)` a una función donde los argumentos son independientes. Haskell provee una función llamada `curry` cuyo tipo es:

```
curry :: ((a, b) -> c) -> a -> b -> c
```

Intuitivamente, se puede pensar que si a una función se le fijan algunos argumentos, el resultado es una función de los argumentos restantes. Esto permite construir nuevas funciones más simples a partir de otras más complejas. Por ejemplo, si se quiere construir una función que, dada una lista de enteros, devuelva una lista con los cuadrados de cada entero, puede usarse una aplicación parcial (o sección, en la jerga) de la función `map`:

```
elevarListaAlCuadrado = map (^2)
```

donde se omite el último parámetro. Es interesante ver el tipo de esta función:

```
elevarListaAlCuadrado :: [Integer] -> [Integer]
```

10.7. Expresiones Lambda

Ha llegado el momento de mostrar otra de las insignias de Haskell, las *expresiones o abstracciones Lambda*. Si bien ya existían varios lenguajes funcionales que las incluían, en Haskell juegan un rol más cotidiano. La equivalencia es casi inmediata y sería la siguiente:

- Definición: $\lambda x.\, e \longrightarrow$ `\x -> e`
- Especialización: $(\lambda x.\, e)a \longrightarrow$ `(\x -> e)a`
- Ejemplo: $(\lambda x.\, x + 15)a \longrightarrow$ `\x -> x + 15`
- Ejemplo con más parámetros: $(\lambda x\, y.\, x + y)a \longrightarrow$ `\x y -> x + y`

En este contexto, currificar el último ejemplo sería:
`\x y -> x + y` $\longrightarrow$ `\x -> \y -> x + y`
Esto es, se transforma una función de dos argumentos que devuelve una expresión, a una función que recibe un argumento y devuelve una función que toma un argumento (el restante) y devuelve una expresión.

Uno de los usos más comunes que se le da a estas expresiones, es la de construir funciones anónimas para ser usadas en lugares muy específicos que no justifican la creación de una nueva función con nombre propio. Un ejemplo típico es al usar la función `map`, construir la función a aplicar en el mismo lugar de la llamada:

```
listaDesplazada l = map (\x -> x + 1) l
```

11. *Layout*

Haskell usa una sintaxis de dos dimensiones llamada *layout* que se basa en organizar las declaraciones alineadas en columnas. Se dice también que Haskell tiene una sintaxis *layout-driven*.

Es importante recordar, en primer lugar, que el próximo caracter que le sigue a cualquiera de las palabras reservadas `where`, `let` o `of` es el que determina la columna de inicio para las declaraciones que vaya a contener. En segundo lugar, hay que asegurar que la primer columna de un bloque esté más a la derecha que la primer columna del bloque inmediatamente abarcante, pues de otra manera sería ambiguo). El fin de un bloque se alcanza cuando se encuentra una línea más a la izquierda que la primera línea del bloque en cuestión.

Es altamente recomendable manipular los fuentes de Haskell desde editores con fuentes monoespaciadas para evitar errores de *parsing*.

12. Inferencia de Tipos

Otra de las características que hacen al sistema de tipos de Haskell especial es la capacidad de inferir el tipo de una expresión sin necesidad de que el programador lo especifique. Esto lo logra analizando las operaciones a las que una expresión está sometida.

El sistema asigna el tipo menos general que contenga todas las instancias posibles de la expresión. Por ejemplo, infiere que el tipo de la función `tail` es `[a] -> [a]`. Sin embargo, los tipos `[a] -> [b]`, `a -> [b]`, `[a] -> b`, `a -> b` y hasta inclusive `a` son válidos; pero todos ellos, en mayor o menor medida, son más generales de lo estrictamente necesario. Por el contrario, `[Integer] -> [Integer]` o `[Char] -> [Char]` serían demasiado específicos y no estarían contemplando todos los casos posibles.

La existencia de tipos principales únicos es la característica más importante del sistema de tipos conocido como *Hindley-Milner*, que forma parte de la base de Haskell, ML, Miranda y otros lenguajes.

13. Tipos de datos algebraicos

Los *Tipos de datos algebraicos* son tipos de datos donde cada uno de sus valores son un tipo de dato envuelto en uno de sus constructores. Por ejemplo, se podría construir el tipo de dato `ColorRGB` a partir de 3 enteros de la siguiente manera:

```
data ColorRGB = RGB Int Int Int
```

Aquí, `RGB` juega el rol de (único) constructor que recibe como parámetro a los tipos base a partir de los cuales se generará el tipo de dato algebraico.

Un ejemplo de uso sería la creación de funciones que los generen:

```
rojo     = RGB 255   0   0
verde    = RGB   0 255   0
azul     = RGB   0   0 255
amarillo = RGB   0 255 255
gris n   = RGB   n   n   n
```

Es importante destacar que los constructores no se ejecutan y la única forma de obtener los datos que *envuelven* es mediante *Pattern Matching*. Por ejemplo, si se quisiera obtener la composición de un `ColorRBG`, se podrían construir las siguientes funciones:

```
cuantoRojo, cuantoVerde, cuantoAzul :: ColorRGB -> Int
cuantoRojo  (RGB r g b) = r
cuantoVerde (RGB r g b) = g
cuantoAzul  (RGB r g b) = b
```

También es posible definir un tipo de dato algebraico a partir de múltiples constructores. Por ejemplo, un tipo de dato que modele alumnos y docentes:

```
data Persona = Docente String Integer | Alumno String Integer
```

y una función que mediante *Pattern Matching* defina un comportamiento distinto para cada caso:

```
informacion :: Persona       -> String
informacion    (Docente n i) =  "Docente " ++ n ++ " - id: "     ++ (show i)
informacion    (Alumno n p)  =  "Alumno "  ++ n ++ " - Padron: " ++ (show p)
```

13.1. Campos etiquetados

El ejemplo anterior donde se definió el tipo `Persona`, solo posee dos campos y no presentan dificultad para ser accedidos posicionalmente. Si se utilizara un tipo más completo, por ejemplo el siguiente:

```
type Nombre   = String
type Apellido = String
type Edad     = Integer
```

```
type Padron   = Integer
type Email    = String
type CBC      = Bool

data Alumno = Alumno Nombre Apellido Edad Padron Email CBC

getNombre :: Alumno               -> Nombre
getNombre    (Alumno n _ _ _ _ _) =  n

getEdad   :: Alumno               -> Edad
getEdad      (Alumno _ _ e _ _ _) =  e

getEmail  :: Alumno               -> Email
getEmail     (Alumno _ _ _ _ e _) =  e
```

se observa como aumenta la posibilidad de confundir el orden de los campos a medida que aumentan en cantidad.

Para evitar esto, Haskell permite etiquetar los campos para luego poder acceder a ellos por su nombre. El ejemplo anterio, podría reescribirse de la siguiente manera:

```
type Nombre   = String
type Apellido = String
type Edad     = Integer
type Padron   = Integer
type Email    = String
type CBC      = Bool

data Alumno = Alumno {nombre   :: Nombre,
                      apellido :: Apellido,
                      edad     :: Edad,
                      email    :: Email,
                      padron   :: Padron,
                      cbc      :: CBC}

getNombre :: Alumno -> Nombre
getNombre    a      =  nombre a

getEdad   :: Alumno -> Edad
getEdad      a      =  edad a

getEmail  :: Alumno -> Email
getEmail     a      =  email a
```

donde cada campo tiene un nombre, y además se han definido, implícitamente, los siguientes *selectores*:

```
nombre   :: Alumno -> Nombre
apellido :: Alumno -> Apellido
edad     :: Alumno -> Edad
email    :: Alumno -> Email
padron   :: Alumno -> Padron
cbc      :: Alumno -> CBC
```

El ejemplo siguiente ilustra como construir nuevos valores, utilizando campos etiquetados:

```
francisco = Alumno {nombre   = "Francisco",
                    apellido = "Albani",
                    edad     = 22,
                    email    = "falbani@fi.uba.ar",
                    padron   = 84891,
                    cbc      = true}
```

Si se omiten algunos campos, el resto queda indefinido.

Para construir patrones, se usa una sintaxis muy similar a la del constructor. Por ejemplo:

```
getNombre :: Alumno              -> Nombre
getNombre    (Alumno {nombre=n}) =  n

getEdad   :: Alumno              -> Edad
getEdad      (Alumno {edad=e})   =  e

getEmail  :: Alumno              -> Email
getEmail     (Alumno {email=e})  =  e
```

Una función puede utilizar los campos etiquetados de una estructura existente para construir una nueva a partir de ella. En el siguiente ejemplo, se "actualiza"la edad de un alumno:

```
franciscoActualizado = francisco {edad = 1 + edad francisco}
```

Por supuesto que no es una actualización destructiva: simplemente se crea una nueva copia tomando por defecto los valores del dato origen.

Esta forma alternativa de expresar un Tipo Algebraico no altera su naturaleza. Es una vía para hacerlos más manejables y extendibles, dado que de esta manera se pueden agregar o quitar nuevos campos sin necesidad de actualizar todas las referencias al constructor.

13.2. Enumerados

Un caso especial son los tipos de datos algebraicos cuyos constructores no envuelven ningún tipo. Su uso principal es para construir enumerados. En el

siguiente ejemplo se pretende nuclear a todos los continentes bajo un mismo tipo[15]:

```
data Continente = America | Europa | Africa | Asia | Oceania
```

Una función que podría hacer uso de Continente sería, por ejemplo, `conectadosTierra`, que, dados dos continentes, informa si estos están o no conectados por tierra:

```
conectadosTierra :: Continente -> Continente -> Bool
conectadosTierra    America       _          =  False
conectadosTierra    Oceania       _          =  False
conectadosTierra    Europa        Asia       =  True
conectadosTierra    Europa        _          =  False
conectadosTierra    Asia          Africa     =  True
conectadosTierra    Asia          Europa     =  True
conectadosTierra    Asia          _          =  False
conectadosTierra    Africa        Asia       =  True
conectadosTierra    Africa        _          =  False
```

13.3. Recursivos

Dada la definición de tipo de dato algebraico en la Sección 13, nada impide utilizar como parámetro de un constructor, al propio tipo que se está definiendo. Esta propiedad recursiva, permite concebir estructuras virtualmente infinitas. Mediante esta técnica podemos representar al clásico ejemplo de los árboles binarios definidos como, o bien vacíos, o constituidos por un nodo que envuelve un dato y dos ramificaciones:

```
data ArbolBinario = ArbolVacio | Nodo Int ArbolBinario ArbolBinario
```

La siguiente función recursiva, dado un árbol, devuelve su profundidad:

```
profundidad :: ArbolBinario -> Integer
profundidad    ArbolVacio   =  0
profundidad    (Nodo l r)   =  1 + max (profundidad l) (profundidad r)
```

13.4. Polimórficos

En muchos casos, el concepto que se está abstrayendo resulta ser una descripción general que envuelve a varios tipos distintos. El ejemplo por excelencia son las Listas, dónde prácticamente todas las operaciones que se realizan sobre las mismas no imponen condiciones sobre el tipo a contener. Vale la pena decir que también son el ejemplo por excelencia de las estructuras recursivas potencialmente infinitas (Sección 13.3). La definición más común es:

```
data Lista a = ListaVacia | Cons a (Lista a)
```

Otro ejemplo muy común son los árboles binarios genéricos:

[15] Se ha dejado de lado el rigor geográfico para simplificar el ejemplo.

```
data ArbolBinario a = ArbolVacio | Nodo a (ArbolBinario a) (ArbolBinario a)
```

Es oportuno destacar que, al ser las listas el tipo de dato de uso más natural en Haskell, se provee de azúcar sintáctico para facilitar su creación y manipulación. A continuación se muestran las equivalencias con la definición anterior:

- `Lista a` $\longrightarrow$ `[a]`
- `ListaVacia` $\longrightarrow$ `[]`
- `Cons a (Lista a)` $\longrightarrow$ `a:[a]`

14. Clases de tipos

La inclusión de Clases de tipos en el diseñoo de Haskell fue una de las decisiones más acertadas del comité, ya que es considerada una de sus características más distintivas. Fueron concebidas por Philip Wadler en 1988 durante una conversación con Joe Fasel acerca de como la sobrecarga debía estar reflejada en el tipo de una función. Wadler entendió erróneamente la idea de Fasel, y ese fue el momento en el que las *Clases de tipos* nacieron. Tiempo después, Steven Blott, alumno de Wadler, ayudó a formular las reglas y a probar la coherencia, sanidad y completitud del sistema, para su tesis doctoral. El comité aceptó la idea sin demasiado debate, en contradicción directa con el objetivo de solo incluir características muy bien probadas y que gozaran de gran consenso.

La motivación detrás de la creación de las Clases de tipos era solucionar un importante problema: sobrecarga de operadores para operaciones numéricas y equivalencia[16]. Para ilustrar lo anterior, se puede considerar la función `elem`, que sirve para saber si una lista contiene a un elemento dado. Su definición es la siguiente:

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

A partir de esto, se esperaría que el tipo de elem sea `a -> [a] -> Bool`. Pero eso implicaría que el tipo de `(==)` sea `a -> a -> Bool`, inclusive cuando hay tipos para los cuales el concepto de igualdad no está bien definido, y es voluntad del programador que sobre ellos no esté definido el operador `(==)`. Más allá de esto, aún suponiendo que el concepto de igualdad estuviera bien definido para todos los tipos, su cómputo depende de cada uno en particular y debería existir un mecanismo para especificarlo.

14.1. Polimorfismo *Ad-Hoc*

Las *Clases de tipo* son el mecanismo que Haskell provee para soportar el Polimorfismo *Ad-Hoc*, más conocido como *sobrecarga*. Este concepto permite que el comportamiento de una misma operación pueda ser diferente para cada tipo de dato distinto. Permite definir un conjunto de operaciones y agruparlas bajo un nombre de *clase* del que luego se especificará que tipos son instancias de la misma. En otras palabras, una clase define un conjunto de reglas que un tipo debe cumplir para considerarse instancia de esa clase.

El ejemplo por excelencia es la clase que define a los tipos de datos cuyos elementos pueden o no ser equivalentes. Su nombre es `Eq` y a continuación se muestra su definición[17]:

[16] Con el correr del tiempo, el concepto se fue generalizando hasta convertir a Haskell en una especie de laboratorio en el cual numerosas extensiones al sistemas de tipos han sido diseñadas e implementadas.

[17] En realidad, la definición es un poco más completa, pero a los efectos de simplificar la explicación, se han omitido algunos detalles.

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

Lo anterior se puede leer de la siguiente forma: *Para cada tipo* `a` *que sea una instancia de la clase* `Eq` *,* `(==)` *es de tipo* `a -> a -> Bool`. Simbólicamente:

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

Se puede ver que ahora, queda explícito en el tipo de la función la restricción a la que sus parámetros son sometidos.

Ahora se podría, por ejemplo, hacer que el tipo de dato `Arbol`, sea una instancia de `Eq` de la siguiente manera:

```
instance (Eq a) => Eq (Arbol a) where
  Hoja a       == Hoja b       = (a == b)
  (Rama l1 r1) == (Rama l2 r2) = (l1==l2) && (r1==r2)
  _            == _            = False
```

Que se lee: *Para cada tipo* `a` *que sea una instancia de la clase* `Eq`*, el tipo* `Arbol a` *es también una instancia de la clase* `Eq`*, a partir de la definición dada del operador* `(==)`. La definición de `(==)` es llamada también **método**. Es importante notar la restricción extra que se impone sobre el tipo `a`. Esencialmente dice que la comparación de igualdad entre árboles de elementos de tipo `a`, solo será posible si estos últimos admiten tal comparación. Estas restricciones se conocen como **Contexto**. Si se omiten, pueden surgir errores de tipado estático.

Es posible también, en la definición de una clase, establecer métodos por defecto. Por ejemplo, una definición más completa de la clase `Eq` es:

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

donde se agrega una operación de desigualdad con un comportamiento por defecto. Si una declaración de instancia de tipo omite la definición de `(/=)`, esta será interpretada como la negación de `(==)`.

14.2. Extensión de clases

Haskell también soporta la noción de extensión de clases. Por ejemplo, se podría definir una clase de tipo ordinal, `Ord`, que herede todas las operaciones de la clase `Eq`, y agregue un conjunto de operaciones que permitan expresar los conceptos de mayor, menor, mínimo y máximo. He aquí su definición:

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

Se puede interpretar como que `Eq` es una superclase de `Ord` o como que `Ord` es una subclase de `Eq`. Cualquier tipo que sea una instancia de `Ord` debe ser también una instancia de `Eq`.

Los métodos de una subclase pueden asumir la existencia de métodos de una superclase. Por ejemplo, una declaración más completa de `Ord` podría contener este método por defecto para `(<)`:

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

donde se supone la existencia de `(/=)`.

A partir de esto, se puede ver que el tipo de la función `quicksort`:

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

refleja explícitamente que solo puede operar sobre listas de elementos ordinales.

14.3. Instancias derivadas

Escribir instancias de `Eq`, `Ord` y varias clases más es en general algo obvio y puede resultar molesto. Afortunadamente, Haskell ofrece un mecanismo para automatizar esta tarea. Si el tipo de dato es lo suficientemente simple, el compilador puede derivar automáticamente varias de las clases mas comunes. Un ejemplo sería:

```
data Alumno = Alumno {nombre   :: Nombre,
                      apellido :: Apellido,
                      edad     :: Edad,
                      email    :: Email,
                      padron   :: Padron,
                      cbc      :: CBC} deriving Eq
```

donde se especifica que el tipo `Alumno` debe ser instancia de la clase `Eq`, mediante la orden `deriving`.

14.4. Comparación con otros lenguajes

- Los métodos definidos en una clase de Haskell corresponden a funciones virtuales en una clase de **C++**. Cada instancia de la clase provee su propia definición para cada método.
- Las clases de Haskell son ligeramente similares a las interfaces de **Java** en el sentido siguiente: se define un protocolo de uso en lugar de definir su implementación.
- Haskell no soporta el estilo de sobrecarga de **C++** en donde funciones con argumentos de distintos tipos pueden compartir el mismo nombre.
- En Haskell no hay una clase universal base como por ejemplo `Object` en Java.

- En Haskell no hay control de acceso, como por ejemplo los modificadores `public` o `private`. En lugar de eso, se debe usar el Sistema de Módulos para ocultar los componentes de una clase.

15. Las mónadas como una clase de tipo

Las mónadas en Haskell son una clase de tipo como cualquier otra, cuya definición es la siguiente:

```
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
```

Donde `(>>)` es una versión más simple de `(>>=)` donde el resultado de la primer acción no es necesitado por la segunda y por eso es descartado.

Al instanciarla se debe garantizar el cumplimiento de las leyes expuestas en la Sección 8.

A parte de esas tres leyes, algunas mónadas también obedecen leyes de adición. Tienen un valor especial llamado `mzero` y un operador `mplus` que obedecen lo siguiente:

- $mzero >>= f \equiv mzero$
- $m >>= (\lambda x \rightarrow mzero) \equiv mzero$
- $mzero \text{`}mplus\text{`} m \equiv m$
- $m \text{`}mplus\text{`} mzero \equiv m$

Para representar a las mónadas con propiedades de adición, Haskell provee una clase derivada de `Monad`:

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

15.1. Las mónadas más ilustres

El Preludio[18] de Haskell incluye varias instancias de la clase `Monad`. A continuación se hará un repaso de las más ilustres:

15.1.1. *Maybe*

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

Este tipo encapsula un valor opcional. Un valor de tipo `Maybe a` o bien contiene un valor de tipo `a`, representado por el constructor `Just a`, o está vacío, representado por el constructor `Nothing`. Es útil para trabajar en situaciones donde una operación puede resultar inconsistente, sin necesidad de tomar medidas drásticas como levantar un error.

[18] Se conoce como *Prelude* al paquete básico de funciones de Haskell.

Cómo una mónada, puede ser vista como una estrategia para secuenciar acciones asociadas a un tipo que pueden fallar y no devolver nada. Su instanciación es la siguiente:

```
instance Monad Maybe where
  return         = Just
  fail           = Nothing
  Nothing  >>= f = Nothing
  (Just x) >>= f = f x

instance MonadPlus Maybe where
  mzero             = Nothing
  Nothing `mplus` x = x
  x       `mplus` _ = x
```

15.1.2. Las listas también son mónadas

La mónada Lista representa la estrategia de combinar una cadena de cómputos no-determinísticos aplicando operaciones a todos los posibles valores en cada paso. Es muy útil para resolver ambigüedades. Su instanciación es la siguiente:

```
instance Monad [] where
  m >>= f  = concatMap f m
  return x = [x]
  fail s   = []

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

15.1.2.1. Comprensión de Listas Otro de los lugares donde Haskell muestra su elegancia matemática es en lo que se conoce como *List Comprehension*, azúcar sintáctico de una construcción monádica que permite generar y expresar conjuntos de una forma muy similar a como se lo hace en la matemática. A continuación, se listas varios ejemplos significativos:

- Los enteros comprendidos entre 1 y 10: `[1..10]`

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

- Los enteros positivos: `[1..]`

  ```
  [1,2,3,4,5,6,7,8,9,10,11,12,13,14...
  ```

- Un conjunto de pares: `[(x, y) | x <- [1..3], y <- [4..5]]`

  ```
  [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
  ```

- Los enteros positivos pares: `[n | n <- [1..], n ‘mod‘ 2 == 0]`

  `[2,4,6,8,10,12,14,16...`

- Las cuadrados de los 15 primeros enteros: `[n*n | n <- [1..15]]`

  `[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]`

- Pares $(x, f(x))$ donde x pertenece al conjunto s: `[(x, f x) | x <- s]`
- $S = \{\, n \mid n \in \mathbb{N}, \sqrt{n} > 100 \,\}$: `S = [ n | n<-[0..], sqrt n>100 ]`.

A partir de los ejemplos anteriores puede verse que a la izquierda del símbolo `|` se expresa la forma de los resultados deseados, y a la derecha se enumeran, separados por coma, los generadores y las *condiciones* que se deben cumplir, conocidas como *guardas*.

Para finalizar, algunos ejemplos clásicos más complejos:

- Lista infinita de números primos utilizando el algoritmo de la *Criba de Eratóstenes*:

```
primos = sieve [2 .. ]
     where sieve [] = []
           sieve (x:xs) = x : sieve [y | y <- xs, y ‘mod‘ x > 0]
```

- Simple implementación del algoritmo de ordenamiento *QuickSort*, muy famosa por ocupar solo 2 líneas:

```
qs [] = []
qs (x:xs) = qs ([c|c <- xs,c <= x]) ++ [x] ++ qs ([c|c <- xs,c > x])
```

- Función para obtener una lista de los números perfectos[19] menores que n:

```
pn n = [p | p <-[1..n], (p == sum [f | f <-[1..(p-1)], (p ‘mod‘ f) == 0])]
```

15.1.3. IO Monad

Esta es sin duda la más significativa pues ha sido la puerta de entrada de las mónadas al mundo de la programación. Su llegada ha traído luz a un sector oscuro de la programación funcional, cual mesías a un pueblo a la deriva.

Como se ha visto en la Sección 8, los lenguajes funcionales puros son incompatibles con la *Entrada/Salida* debido a su opacidad referencial. Esta incompatibilidad se resuelve confinando los cómputos que realicen **I/O** dentro de la mónada. Al no proveerse mecanismos que permitan remover los datos contenidos

[19] Un número perfecto es un entero con la propiedad de ser igual a la suma de los divisores propios menores que él mismo.

por ella, se dice que es una mónada *sin salida*: los efectos colaterales quedan aislados del resto funcional puro del programa. Las *acciones* se van ejecutando a medida que se van ligando, permitiendo secuenciar una serie de cómputos en un estilo que emula muy bien al de la programación imperativa.

En Haskell, la función con la cual se dispara la ejecución, `main`, debe tener el tipo `IO ()`. Es por esto que los programas en Haskell, son típicamente estructurados como una secuencia de acciones **I/O** de estilo imperativo en el nivel más alto, con llamadas al código funcional puro. Las funciones del módulo `IO` no realizan tareas de **I/O** por su cuenta, pero devuelven *acciones **I/O***, que describen la operación deseada. Estas acciones pueden ser combinadas dentro de la mónada `IO` (de una manera funcional pura) pra crear acciones **I/O** más complejas, resultando en la acción **I/O** final que es el valor principal del programa.

La definición de `IO` es dependiente de la plataforma; pero aún así, resulta útil tener una idea de como es su instancia a la clase `Monad`:

```
instance Monad IO where
    return a = ...   -- Función de tipo: a -> IO a
    m >>= k  = ...   -- Ejecuta la acción IO m y liga el valor ..
    fail s   = ...   -- .. que contiene a la entrada de k
```

A continuación, se mostrará un pequeño programa que pide al usuario que ingrese un texto, para luego invertirlo y mostrarlo. Este programa responde a un esquema muy simple de **entrada-procesamiento-salida**. El objetivo es mostrar como, efectivamente, la función `main` es el resultado de la combinación de acciones que, sin salir de la mónada `IO`, interactúan con el exterior para obtener y entregar datos, y con el interior funcional puro para procesarlos:

```
escribirLinea :: [Char] -> IO ()
escribirLinea []        = (putChar '\n') >> return ()
escribirLinea (c:cs)    = (putChar c) >> (escribirLinea cs)

leerLinea :: IO [Char]
leerLinea = getChar >>= \c ->
            if c == '\n' then
                return []
            else
                leerLinea >>= \cs ->
                return (c : cs)

main :: IO ()
main = escribirLinea "Ingrese el texto: " >>
       leerLinea >>= \s ->
       escribirLinea (reverse s)
```

El Preludio de Haskell provee funciones para leer y escribir líneas pero, por cuestiones didácticas se optó por crear nuevas con implementaciones bien explícitas, para enriquecer el ejemplo.

Es notable el aspecto imperativo de la función `main`. La función `reverse` forma parte del Preludio y juega el rol de código funcional puro encargado de procesar los datos.

15.1.3.1. *Do-notation* [20]

Otra conclusión que se cae de madura del ejemplo de la sección anterior, es que los operadores `(>>=)` y `(>>)` junto a las expresiones lambda anónimas se vuelven más y más intrusivas a medida que el código cobra tamaño. Es por eso que Haskell provee azúcar sintáctico para poder escribir las secuencias de cómputos en un estilo imperativo más ordenado, emulando inclusive a la asignación de variables. El resultado de un cómputo monádico puede ser *asignado* a una variable usando el operador `<-`. Luego, al usar esa variable en un cómputo subsecuente, automáticamente se realiza la ligadura. El tipo de la expresión a la derecha de `<-` es monádico, `m a`. La expresión de la izquierda debe ser un patrón para enfrentar con el valor dentro de la mónada.

La traducción de la notación común a la notación *do* indica que cada construcción de la forma `e1 >>= \x ->` puede escribirse como `x <- e1` y cada una de la forma de `e2 >>= \_ ->` se transforma en, simplemente `e2`.

A continuación se expondrá una nueva versión del programa anterior donde se utiliza la notación *do*:

```
escribirLinea :: [Char] -> IO ()
escribirLinea []        = do putChar '\n'
                             return ()
escribirLinea (c:cs)    = do putChar c
                             escribirLinea cs

leerLinea :: IO [Char]
leerLinea = do c <- getChar
               if c == '\n' then return []
                   else do cs <- leerLinea
                           return (c : cs)

main = do escribirLinea "Ingrese el texto: "
          s <- leerLinea
          escribirLinea (reverse s)
```

[20] Si bien la presentación de este concepto está en el contexto de la mónada **IO**, no debe olvidarse que puede utilizarse con cualquier otra.

16. Módulos

Un programa típico escrito en Haskell se forma a partir de la unión de módulos. Este concepto sirve para controlar los espacios de nombres y como mecanismo para crear tipos de datos abstractos (ADT's).

Un módulo puede incluir cualquiera de las declaraciones mostradas anteriormente: funciones, tipos de datos, clases, instancias, sinonimias, etc. El único requisito que existe en cuanto al orden de aparición, es el que obliga a realizar todas las importaciones al principio del módulo.

Las implementaciones más comunes de compiladores de Haskell asocian a cada módulo un archivo. Por ejemplo, un módulo destinado a nuclear funciones de números complejos, estaría contenido en el archivo `Complejos.hs`. Un módulo que quiera hacer uso de él, deberá importarlo al principio con la sentencia `import Complejos`. A continuación, un ejemplo de archivo `Complejos.hs`:

```
module Complejos where

type Complejo = (Float, Float)

conjugado :: Complejo -> Complejo
conjugado (a, b) = (a, b*(-1))

re, im, modulo :: Complejo -> Float
re (a, b) = a
im (a, b) = b
modulo (a, b) = sqrt (a*a + b*b)

suma, mult :: Complejo -> Complejo -> Complejo
suma (a, b) (c, d) = (a+c, b+d)
mult (a, b) (c, d) = (a*c - b*d, a*d + c*b)
```

y un ejemplo de módulo que lo utilice:

```
import IO
import Complejos

main = do putStrLn "Ingrese la parte real: "
          r <- getLine
          putStrLn "Ingrese la parte imaginaria: "
          i <- getLine
          putStrLn ("El modulo de " ++ show r ++ " + " ++ show i ++
          "i es: " ++ show (modulo (4, 3)) ++ ".")
```

En este caso, la llamada `import Complejos` "trae"todas las definiciones de `Complejos.hs` al módulo. Existen mecanismos para limitar esto y pueden estar tanto en el módulo origen como en el módulo destino. En el caso en el que se desee importar solo algunas cosas de un módulo externo, las limitaciones se

establecerán dentro del módulo destino; en el caso en el que se desee restringir el acceso a un módulo (el caso de los ADT's) las limitaciones estarán en el módulo origen.

16.1. Restricción al importar

Para especificarle a `import` específicamente las cosas que tiene que importar, hay que listarlas de la siguiente forma:

```
import Complejos (modulo, suma)
```

Es posible también especificar que cosas **no** exportar, utilizando la orden `hiding`:

```
import Complejos hiding (suma, mult)
```

16.2. Restricción al exportar (ADT's)

La única forma de construir tipos abstractos de datos (ADT's) en Haskell, es utilizando módulos con restricciones a la hora de exportar.

Un ejemplo simple, podría ser el caso de la creación de un ADT Árbol. El contenido del módulo que lo representaría sería:

```
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
```

donde se puede observer como, al momento de declarar el módulo, se omite en la lista de exportación a los dos constructores de `Tree a`.

Parte IV
Conclusión

Aquí concluye la redacción del informe de *Haskell*, lenguaje que me fuera asignado para investigar como asignatura en la materia *Teoría de Lenguajes*, el segundo cuatrimestre del año 2006.

Ha sido el fruto de largas horas de lectura de publicaciones teóricas, de tutoriales prácticos y de ensayos lúdicos. Su realización me abrió las puertas al mundo de los Lenguajes Funcionales, sector de la programación que aún no conocía y motivo de mucha intriga; practicamente todo lo expuesto en este trabajo ha sido para mi nuevo.

Muchas cosas han quedado en el tintero por limitaciones, en parte externas, como la falta de tiempo, y en parte propias, como la incapacidad de comprender suficientemente a fondo ciertos temas tan rápidamente. Hubiera querido presentar algunas otras características de Haskell como por ejemplo *Lazy patterns*, *newtype*, funciones infijas, patrones $n+k$. Tambien me hubiera gustado dar información sobre los compiladores, intérpretes y librerías disponibles; así como también confexionar una galería de aplicaciones famosas escritas en Haskell. Ejemplos de mónadas utilizadas para simular estado, excepciones y concurrencia son quizá las gotas de tinta más importantes que no he derramado.

Aún así, el producto final y el valor de los conocimientos adquiridos en la cursada de la materia y en la investigación me han dejado harto conforme.

Francisco Albani
Buenos Aires,
7 de Marzo de 2007.

Parte V
Bibliografía

Referencias

[Bac78] John Backus.
"Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs" (1978)
`http://doi.acm.org/10.1145/359576.359579`.

[Hug89] John Hughes.
"Why Functional Programming Matters" (1989)
`http://www.math.chalmers.se/~rjmh/Papers/whyfp.html`.

[Pey00] Simon Peyton Jones.
"Tackling the awkward squad" (2000)
`http://research.microsoft.com/~simonpj/papers/history-of-haskell/index.htm`.

[His07] Paul Hudak, John Hughes, Simon Peyton Jones & Philip Wadler.
"A History of Haskell: being lazy with class" (2007)
`http://research.microsoft.com/~simonpj/papers/history-of-haskell/index.htm`.

[Haskellorg] Haskell Community.
"Haskell Official Site"
`http://www.haskell.org`.

[HaskellIrc] Haskell IRC Users.
"Haskell IRC"
`http://haskell.org/haskellwiki/IRC_channel`.

[Gen00] Paul Hudak, John Peterson & Joseph Fasel.
"A Gentle Introduction to Haskell"
`http://www.haskell.org/tutorial/`.

[Ros06] Andrea Rossato.
"The Monadic Way" (2006)
`http://www.haskell.org/haskellwiki/The_Monadic_Way`.

[Dau02] Hal Daumé III.
"Yet Another Haskell Tutorial" (2002)
`http://www.cs.utah.edu/~hal/htut/`.

[New] Jeff Newbern.
"All about Monads"
`http://www.haskell.org/all_about_monads/html/index.html`.