Paradigmas de Lenguajes de Programación
Introducción al paradigma funcional

Introducción (a la Introducción)

El lenguaje Haskell entra en el paradigma de **programación funcional**, que consiste, básicamente, en pensar que un programa es una función, un aparato que transforma datos de entrada en un resultado. Los **lenguajes funcionales** nos dan herramientas para explicarle a la computadora cómo calcular esa función.

Una de las herramientas que nos brindan son las **expresiones**. Las expresiones son una tira de símbolos que representan (**denotan**) un **valor**. Por ejemplo, las expresiones `2`, `1+1`, `(3*7+1)/11` representan todas el mismo valor.

Los valores se agrupan en **tipos**. Por ejemplo, podemos decir que el valor representado por las expresiones del párrafo anterior pertenece al tipo **número entero**. Hay un tipo de datos distinto que contiene ciertos números racionales y se llama **número de punto flotante** por la forma en que se representa en la computadora. Otro tipo de datos es el tipo **valor de verdad** que está formado por solamente dos valores: verdadero y falso. Y hay un tipo que tiene todos los **caracteres** (letras y símbolos). En Haskell estos tipos se llaman `Int`, `Float`, `Bool` y `Char`, respectivamente. También hay tipos que se forman a partir de

otros, como el tipo que contiene todos los pares ordenados de un entero y un valor de verdad.

Dije que una expresión representa un valor, y una propiedad muy importante de la programación funcional es lo que se llama **transparencia referencial**: cada expresión representa siempre el mismo valor, sin importar en qué lugar de un programa aparece. Otros paradigmas permiten que una misma expresión o instrucción represente un valor o indique una acción distinta, de acuerdo al contexto en el que se encuentre. Pero la programación funcional, no.

La transparencia referencial es muy útil a la hora de modificar un programa, ya que no tenemos que preocuparnos porque las modificaciones que hagamos en una parte del mismo afecten los cálculos que se hacen en otras. También es una aliada muy poderosa a la hora de verificar un programa (demostrar matemáticamente que cumple la especificación), dado que podemos utilizar propiedades ya demostradas de todas las subexpresiones que constituyen una expresión, sabiendo que valen en cualquier contexto.

Las expresiones más simples se llaman **expresiones atómicas**. También se las llama **formas normales**, porque son la manera intuitiva de representar un valor. Ejemplos:

- `2`
- `False`
- `(3, True)`

Es común llamarlas “valores” (aunque, en realidad, no *son* un valor, sino que *representan* un valor, como el resto de las expresiones).

También hay **expresiones compuestas** que se construyen combinando expresiones atómicas con operaciones:

- `1+1`
- `1==2`
- `(4-1, True || False)`

Algunas cadenas de símbolos no forman expresiones (y no representan ningún valor). Se las llama **expresiones mal formadas**. Pueden serlo por un problema sintáctico:

- `+*1-`
- `(True`
- `('a',)`

o por un error de tipos:

- `2 + False`
- `2 || 'a'`
- `4 * 'b'`

Para saber si una expresión está bien formada tenemos que aplicar **reglas sintácticas** y **reglas de asignación de tipos** (o de **inferencia de tipos**).

Una particularidad de la programación funcional es que, como en matemática, las **funciones** son elementos (valores). Es así que tenemos dos formas de entenderlas. Según la **visión denotacional**, una función es un objeto matemático que relaciona cada elemento de un conjunto (de partida) con un único elemento de otro (de llegada). Según la **visión operacional**, una función es un mecanismo que, dado un elemento del conjunto de partida calcula (o devuelve) un elemento del conjunto de llegada (su **resultado**).

Veámoslo en un ejemplo:

```
doble x = x + x
```

Según la visión denotacional, esto define un conjunto de pares, en el que cada número está asociado con su doble: {(0,0), (1,2), (2,4), (3,6), ...}. Según la visión operacional, `doble` es una operación que toma un número como argumento y devuelve ese número sumado consigo mismo: `doble 0` → `0`, `doble 1` → `2`, `doble 3` → `6`, etc.

Juntando las dos visiones, la función es un valor y la operación básica que podemos realizar con ese valor es la **aplicación**: aplicar la función a un elemento para obtener un resultado. Sintácticamente, la aplicación se escribe como una yuxtaposición (simplemente se escribe la función seguida de su parámetro). Si `f` es una función y `e` un elemento de su conjunto de partida, entonces `f e` denota el elemento que se relaciona con `e` por medio de la función `f`. Por ejemplo, `doble 2` representa al número `4`.

Dada una expresión bien formada, ¿cómo hacemos para determinar cuál es el valor que denota? Usamos **ecuaciones**, como por ejemplo la que escribí arriba para definir la función `doble`. Partiendo de una expresión compuesta, reemplazamos cada una de sus subexpresiones por otras usando las ecuaciones que tenemos.

Pero este procedimiento podría no terminarse nunca, aun si las ecuaciones están bien pensadas. Por ejemplo, para evaluar la expresión `doble (1 + 1)`, yo podría empezar cambiando la subexpresión `(1 + 1)` por `doble 1`, según la ecuación de definición de `doble`, con lo que obtengo `doble (doble 1)`. Ahora, uso nuevamente la misma ecuación reemplazando `doble 1` por `1 + 1` y vuelvo a empezar desde el punto de partida. No calculé ningún valor.

Para evitar este problema, decimos que en programación funcional usamos **ecuaciones orientadas**. Son ecuaciones en las que el lado izquierdo se considera una expresión a definir y el lado derecho su definición (una expresión ya definida o definida en otra parte). Entonces el cálculo se lleva a cabo reemplazando siempre subexpresiones que formen el lado izquierdo de una ecuación por otras que estén del lado derecho. En este caso, `doble (1 + 1)` pasará a ser `(1 + 1) + (1 + 1)` y esto seguramente, por las ecuaciones de definición de la suma, se transformará primero en `2 + (1 + 1)`, después en `2 + 2` y, finalmente, en `4`.

Este proceso, de reemplazar una subexpresión por otra que la define, sin tocar el resto de la expresión, se llama una **reducción**. La expresión resultante no necesariamente es más corta, pero seguramente está “más definida”, más cerca de ser una forma normal (un valor).

Podemos aplicar las dos visiones a una ecuación orientada de la forma $e_1 = e_2$. En la visión denotacional, estamos definiendo que el valor denotado por e_1 (su significado) es el mismo que el denotado por e_2. En la visión operacional, la ecuación indica que para calcular una expresión que contiene a e_1, reemplacemos todas sus apariciones por e_2.

Ya están en condiciones de entender qué es un **programa funcional** (también llamado un **script**). Es un conjunto de ecuaciones que definen una o

más funciones. En realidad, definen valores en general; pero las funciones son un caso particular ¡y el más interesante!

¿Para qué se usa un programa funcional? Simplemente para reducir expresiones. A primera vista puede no ser tan claro que esto resuelva un problema, pero iremos viendo que sí puede hacerlo. De hecho, las ecuaciones orientadas, junto con el mecanismo de reducción describen algoritmos, pasos para resolver un problema.

Veamos unos ejemplos de funciones:

- `doble x = x+x`
- `fst (x,y) = x`
- `distancia (x,y) = sqrt (x^2+y^2)`
- ```
  signo 0 = 0
  signo x | x > 0 = 1
  signo x | x < 0 = -1
  ```
- `promedio1 (x,y) = (x+y)/2`
- `promedio2 x y = (x+y)/2`
- ```
  fact 0 = 1
  fact n | n > 0 = n * fact (n-1)
  ```
- ```
  fib 0 = 1
  fib 1 = 1
  fib n | n > 1 = fib (n-1) + fib (n-2)
  ```

Tipos de datos

Un **tipo de datos** es un conjunto de valores a los que se les pueden aplicar las mismas operaciones. En la sección anterior mencioné algunos ejemplos de tipos como enteros, punto flotante, valores de verdad, caracteres. También mostré tipos que se formaban a partir de otros, como los pares ordenados con enteros de primera componente y caracteres de segunda.

En Haskell, todo valor pertenece a algún tipo. Dado que las funciones son valores para los lenguajes funcionales, también tienen un tipo; por ejemplo, el tipo “funciones de enteros en enteros”.

Si todos los valores pertenecen a un tipo y toda expresión bien formada denota un valor, queda claro que a cada expresión tiene que corresponderle un tipo (el tipo del valor denotado por ella), es lo que se llama el **tipo de una expresión**.

Haskell es un lenguaje **fuertemente tipado**. Esta propiedad consiste en no permitir que se usen elementos de un tipo para realizar operaciones que esperan argumentos de otro. Gracias a esta restricción, el intérprete puede asegurarse de que estemos “haciendo las cosas bien” en lo que a tipos se refiere. También posee **tipeo estático**: no hace falta hacer “funcionar” un programa para saber si tiene o no un error de tipos.

Esto se hace gracias a un conjunto de reglas llamadas **reglas de inferencia de tipos**. Son reglas que permiten, dada una expresión, averiguar a qué tipo pertenece. Vamos a ver en seguida unos ejemplos de estas reglas, lo
  ```

único que necesitan saber antes es un detalle de notación: `e :: T` se lee "la expresión `e` es de tipo `T`." O sea que el valor denotado por `e` pertenece al conjunto de valores llamado `T`. Ejemplos:

- `2 :: Int`
- `False :: Bool`
- `'a' :: Char`
- `doble :: Int -> Int`

Esta notación nos va a servir para describir las reglas y razonar sobre Haskell, pero también se puede emplear *dentro* de los programas Haskell. Se usa para indicar de qué tipo queremos que sean los nombres que definimos. Así el intérprete chequea que realmente el tipo coincida con el de las expresiones que los definen. Podemos no hacerlo, pero nos perdemos la oportunidad del doble chequeo (el nuestro y el del intérprete), por eso se considera buena práctica de programación. Además, existen casos en los que sí es obligatorio indicar el tipo de lo que estamos definiendo, para dirimir ambigüedades.

Visto esto, ya podemos pasar a algunos ejemplos de reglas de inferencia de tipos:

1. `(e1+e2) :: Int`, si y solo si `e1, e2 :: Int`
2. `f :: T1 -> T2` y `e :: T1`, si y solo si `f e :: T2`
3. `d = e` y `e :: T`, entonces `d :: T`

La primera dice que la suma de enteros es entera. La segunda concierne a la aplicación de funciones: tengo una función que recibe argumentos de tipo `T1` y da resultados de tipo `T2`; cuando aplico la función a una expresión de tipo `T1`, estoy construyendo una expresión (`f e`) de tipo `T2`. Según la tercera, los dos lados de una ecuación tienen que ser del mismo tipo.

Probemos ahora aplicar estas reglas en un ejemplo práctico: ¿de qué tipo es la función `doble`, definida por `doble x = x+x`?

a. Supongamos `(x+x) :: Int` (esto debería indicarlo el programa)
b. Por regla 1, `x :: Int`
c. Por regla 3, `doble x :: Int`
d. Por regla 2, `doble :: Int -> Int`

El sistema de tipos de Haskell está basado en un desarrollo teórico que es el **sistema de tipos de Hindley-Milner**. El sistema define ciertos tipos de datos y reglas de inferencia. Ya vimos algunos **tipos básicos**, como `Int`, `Float`, `Bool` y `Char`. También hay formas de construir **tipos compuestos**:

- **Tuplas**. Si `T1` y `T2` son tipos (simples o compuestos), entonces `(T1,T2)` es otro tipo de datos. Se pueden formar tuplas con cualquier cantidad de componentes. Las expresiones se construyen igual que el nombre del tipo, poniendo las subexpresiones entre paréntesis, separadas por comas.

- **Listas**. Si `T` es un tipo, `[T]` es otro. Más adelante vamos a ver qué significado tienen las listas y cómo se escriben expresiones de estos tipos.
- **Funciones**. Si `T1` y `T2` son tipos, `T1->T2` es también un tipo, el tipo de las funciones de `T1` en `T2`.

Dije que el tipado en Haskell es fuerte. Sin embargo, es habitual querer escribir funciones que puedan usarse, sin tener que redefinirlas, para trabajar con distintos tipos de datos. Esto es lo que se llama **polimorfismo**. Se usa en casos en que el comportamiento de la función no depende del valor de sus parámetros. Un ejemplo muy claro es la función `id` (identidad) definida de la siguiente forma:

```
id x = x
```

¿De qué tipo es `id`? Podemos escribir `id 3` y las reglas que vimos nos indican que `id :: Int -> Int`. También podemos escribir `id True`, y veremos que `id :: Bool -> Bool`; o `id doble`, y entonces será `id :: (Int -> Int) -> (Int -> Int)`.

La idea es que `id :: a -> a` sin importar qué tipo sea `a`. Y esto podemos decirlo en Haskell usando **variables de tipo**. Lo cierto es que la declaración de tipo que aparece al principio de este párrafo (`id :: a -> a`) es válida en Haskell. Se lee "`id` es una función que dado un elemento de algún tipo `a` devuelve otro elemento de ese mismo tipo." Indica que el de `id` es un **tipo parámetrico** (depende de un parámetro) y que, por lo tanto, `id` es una **función polimórfica**.

Como ya dije, es una buena práctica declarar el tipo de los nombres que definimos en un programa para que el intérprete verifique que la definición responde al tipo declarado. Antes definí algunas funciones. Ahora voy a completar la definición, agregándoles antes la declaración de tipo:

- ```
  doble :: Int -> Int
  doble x = x+x
  ```
- ```
  fst :: (a,b) -> a
  fst (x,y) = x
  ```
- ```
  distancia :: (Float,Float) -> Float
  distancia (x,y) = sqrt (x^2 + y^2)
  ```
- ```
  signo :: Int -> Int
  signo 0 = 0
  signo x | x > 0 = 1
  signo x | x < 0 = -1
  ```
- ```
  promedio1 :: (Float,Float) -> Float
  promedio1 (x,y) = (x+y)/2
  ```
- ```
  promedio2 :: Float -> Float -> Float
  promedio2 x y = (x+y)/2
  ```
- ```
  fact, fib :: Int -> Int
  fact 0 = 1
  fact x = x * fact (x-1)
  ```
  ```

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

Presten atención a la diferencia entre `promedio1` y `promedio2`. Radica en el tipo de datos de las funciones. La primera recibe como único argumento un par ordenado. La segunda recibe dos argumentos de tipo `Float`. Cuando declaramos el tipo de la función, separamos los tipos de los dos argumentos por una flecha, igual que la que separa los tipos de los argumentos del tipo del resultado. Esta no es una decisión caprichosa, tiene varias implicancias teóricas y prácticas (que vamos a ver en clase). La notación (en realidad, una de las operaciones que permite) se llama **currificación**, en honor al matemático estadounidense Haskell B. Curry (de cuyo nombre toma también el suyo el lenguaje que están aprendiendo). Por el momento, es suficiente notar que evita el uso de varios signos de puntuación (paréntesis y comas), nótenlo por ejemplo comparando estas expresiones:

```
promedio1 (promedio1 (2, 3), promedio1 (1, 2))
promedio2 (promedio2 2 3) (promedio2 1 2)
```

Para que podamos utilizar un tipo de datos, debe tener, además de un conjunto de *elementos* o valores, un conjunto de *operaciones* que nos permita manipular esos elementos. Todos los tipos de datos con los que estuvimos trabajando están a nuestra disposición siempre que programamos en Haskell. Pero también existe la posibilidad de definir nuevos tipos y ponerles nombre, para usarlos en nuestros programas.

Cada uno de los tipos que utilizamos nos puede haber sido entregado de dos maneras distintas: como un **tipo algebraico** o como un **tipo abstracto**.

Cuando empleamos un tipo algebraico, podemos conocer la *forma* que tiene cada elemento. Esto quiere decir que contamos con un mecanismo para inspeccionar cómo está construido cada dato.

En cambio, de los tipos abstractos solamente conocemos una serie de operaciones. No sabemos cómo están formados los elementos. Y la única manera que tenemos de obtener información sobre ellos es mediante las operaciones.

Para aclarar este punto, veamos cómo podemos hacer para definir nuestros propios tipos algebraicos (para ser usados por nosotros mismos o por otros programadores).

Tipos algebraicos

Para crear un tipo algebraico, tenemos que decir qué forma va a tener cada elemento. Lo hacemos definiendo *constantes* que se llaman **constructores**. Los nombres de los constructores, al igual que los nombres de los tipos, deben empezar con mayúscula. Si bien los constructores pueden tener argumentos, no hay que confundirlos con funciones, ya que no tienen reglas de inferencia asociada. Forman expresiones atómicas (valores). Un ejemplo muy sencillo de tipo algebraico es el tipo `Bool`. Es un tipo con dos constructores:

```
True :: Bool
False :: Bool.
```
```

Las **cláusulas de definición de tipos algebraicos** empiezan con la palabra `data` y tienen la doble función de definir el tipo y sus constructores. Cada constructor da una alternativa distinta para construir un elemento del tipo. Los constructores se separan entre sí por barras verticales. Veamos dos ejemplos:

- `data Sensacion = Frio | Calor`
- `data Forma = Circ Float | Rect Float Float`

El primer tipo tiene dos constructores sin parámetros. Por lo tanto, el tipo `Sensacion` tiene únicamente dos elementos, como el tipo `Bool`. El segundo es más interesante, hay dos constructores posibles, pero los constructores tienen parámetros. Esto quiere decir que algunas figuras son círculos y otras rectángulos; los círculos se diferencian por un número (su radio) y los rectángulos por dos (su base y su altura). Veamos ejemplos de cómo usar este tipo:

- `c1 = Circ 1`
- `c2 = Circ (4.5-3.5)`
- `circulo x = Circ (x+1)`
- `r1 = Rect 2.5 3`
- `cuadrado x = Rect x x`

Las funciones `circulo` y `cuadrado` crean elementos del tipo usando los constructores.

Hablé de un mecanismo para ver cómo estaba construido un elemento de un tipo de algebraico. Me refería a que, si definimos una función que recibe como parámetro una `Forma`, vamos a poder fijarnos si la forma es un círculo o un rectángulo y, en ambos casos, con qué parámetros fue construida. Esta forma de acceso se llama **pattern matching** (algo así como “correspondencia” o “coincidencia de patrones”).

Los **patterns** son expresiones del lenguaje formadas solamente por constructores y variables (argumentos de los constructores) que no se repiten. O sea que `Rect x y` es un patrón, pero `3 + x` no. ¡Y tampoco `Rect x x`, que tiene una variable repetida!

Matching es una operación asociada a un pattern. Lo que hace es tomar una expresión cualquiera y ver si su valor coincide por su forma con el pattern (tal vez se necesite reducirla, evaluarla, antes un poco). Si la correspondencia existe, entonces liga las variables del pattern a las subexpresiones correspondientes. Veámoslo en un ejemplo:

```
area :: Forma -> Float
area (Circ radio) = pi * radio^2
area (Rect base altura) = base * altura
```

En el lado izquierdo de cada ecuación aparece la función que estamos definiendo aplicada a un pattern (`Circ radio` y `Rect base altura` son patterns: un constructor aplicado a variables sin repetir).

Si evaluamos la expresión `area (circulo 2)`, el intérprete debe elegir cuál de las ecuaciones de `area` utilizar. Necesita primero evaluar `circulo 2` para saber a qué constructor corresponde. La reducción da `Circ (2+1)`. Ahora ya se puede verificar cada ecuación para buscar el matching. Esto se logra con la primera ecuación, por lo que la variable `radio` queda ligada a `(2+1)` y, luego de varias reducciones (aritméticas) más, se llega al valor de la expresión: `28.2743`. Como ejercicio, pueden probar reducir la expresión `area (cuadrado 2.5)`.

Las tuplas también son tipos algebraicos, aunque tienen una sintaxis especial, ya que los paréntesis y las comas cumplen la función de constructores. Sin que lo supieran, ya usamos el pattern matching sobre tuplas en ejemplos anteriores. Veámoslos (agregando uno):

- ```
  fst :: (a,b) -> a
  fst (x,y)=x
  ```
- ```
  snd:: (a,b) -> b
  snd (x,y)=y
  ```
- ```
  distancia :: (Float,Float) -> Float
  distancia (x,y) = sqrt (x^2 + y^2)
  ```

También podríamos haber definido `distancia` sin usar pattern matching:

```
distancia p = sqrt ((fst p)^2 + (snd p)^2)
```

Aunque para hacerlo recurrimos a dos funciones que sí habían sido definidas por pattern matching.

Solamente deberíamos optar por usar tipos algebraicos (en lugar de abstractos) cuando se cumplen dos condiciones:

1. Todo valor armado con un constructor y con valores cualesquiera para sus parámetros es un elemento válido para el tipo (y no hay otros).
2. Dos elementos del tipo son iguales entre sí siempre y cuando estén construidos usando los mismos constructores aplicados a los mismos valores.

Estas son condiciones ideales. A veces se construyen tipos algebraicos sin respetarlas del todo. Entonces hay que tener cuidado al definir las funciones que los usan. Sin ir más lejos, el tipo `Forma` con el que estuvimos trabajando no cumple la primera condición, ya que `Circ (-1.2)` no es una forma válida. Y tampoco cumple la segunda, porque interpretando la igualdad como congruencia entre formas (que es la que tiene sentido aquí), `Rect 2.3 4.5` debería ser igual a `Rect 4.5 2.3` y no lo es (no están construidos de la misma manera).

Una buena aplicación para los tipos algebraicos es la de los números complejos (siempre usando `Float` para representar los reales). Veamos que cumplen las dos condiciones:

1. Toda combinación de dos `Float` es un complejo (y estos son todos).
2. Dos complejos son iguales si y solamente si tienen iguales la parte real y la imaginaria.
```

Entonces definamos

```
data Complejo = C Float Float
```

Tenemos un único constructor (la letra `C`) con dos parámetros. No hay barras verticales, porque no hay más de una forma de construir un complejo. ¡No crean que siempre que aparece la palabra `data` debe haber barras verticales! Ese es un error común entre quienes empiezan a aprender estos lenguajes.

Definamos ahora tres operaciones para los complejos, dos de ellas usando pattern matching:

```
parteReal, ParteImag :: Complejo -> Float
ParteReal (C r i) = r
ParteImag (C r i) = i

hacerPolar :: Float -> Float -> Complejo
hacerPolar rho theta = C (rho * cos theta) (rho * sin theta)
```

¿Por qué no usé pattern matching para la tercera? Porque no recibe parámetros de tipo `Complejo`, lo que hace es crear elementos del tipo usando el constructor.

Supongamos que se nos ocurre ahora representar los números racionales como un tipo algebraico construido con dos enteros y definir operaciones para obtener estos enteros a partir de un racional:

```
data Racional = R Int Int

numerador, denominador :: Racional -> Int
numerador (R n d) = n
denominador (R n d) = d
```

¿Les parece buena idea? ¡No debería! Porque este tipo no cumple ninguna de las propiedades vistas:

1. No todo par de enteros es un número racional: `R 1 0`.
2. Hay racionales iguales con distinto numerador y denominador: `R 4 2 = R 2 1`.

Vamos a clasificar los tipos algebraicos en cuatro grupos, lo cual nos va a servir también para tener un panorama más amplio. De cada grupo les doy algunos ejemplos (¡no todos!).

1. **Enumerativos**. Solamente tienen constructores sin parámetros: `Sensacion`, `Bool`.
2. **Productos**. Hay un único constructor con varios argumentos (podría tener uno solo, pero no es un caso muy interesante): `Complejo`, tuplas.
3. **Sumas**. Hay varios constructores con argumentos: `Formas` (y dos tipos de los que vamos a hablar ahora: `Maybe` y `Either`.)
4. **Recursivos**. Utilizan como argumento el mismo tipo definido: listas y árboles, que vamos a ver más adelante.

Veamos los dos tipos de los que les prometí hablar: `Maybe` y `Either`. No los mostré antes, aunque son bastante simples, porque tienen una particularidad: llevan un argumento en el nombre del tipo (no solamente en los constructores).

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

Esto pasa también con las tuplas, pero como los constructores son símbolos les resultó más claro verlo. Lo que quiere decir es que no estamos construyendo un único tipo, sino una familia de tipos: `Maybe Int`, `Maybe Bool`, `Maybe [([Char], Complejo)]`, etc.

Por ejemplo, el tipo `Maybe Int` tiene el elemento `Nothing`, el elemento `Just (-7)`, el elemento `Just 423` y otros. Siempre `Maybe a` es un tipo que tiene un elemento por cada uno del tipo `a` (todos los `Just x`) más un elemento adicional llamado `Nothing`.

`Maybe` se usa para los casos en que queremos convertir en total una función parcial. Se dice que una función `f :: T1 -> T2` es una **función total** si tiene como dominio todos los valores del tipo `T1`. O sea que la función devuelve un valor en `T2` para cualquier expresión correcta de `T1` a la que la apliquemos. Decimos que es una **función parcial**, si tiene como dominio un subconjunto de los valores de `T1`. Quiere decir que podemos escribir expresiones bien formadas con esta función y que estén bien tipadas, pero que no tengan un valor a la hora de calcularlas. Esto pasa con la expresión `1/0`, entre otras.

Al final del apunte vamos a retomar este tema de las funciones parciales y de las funciones totales. Quiero que vean un ejemplo de cómo se puede usar `Maybe` para convertir en total una función (no siempre se puede hacer).

Digamos que queremos hacer una función que busque un dato en una lista. La lista está formada por pares. Cada par tiene una clave de búsqueda y un dato. Por ejemplo, la clave podría ser el DNI, y el dato el nombre completo de la persona. Como no sabemos de qué tipo van a ser la clave y el dato, escribimos una función polimórfica:

```
buscar :: clave -> [(clave,dato)] -> dato
```

No voy a escribir las ecuaciones de la función, porque todavía no estamos trabajando con listas, pero lo que hace la función es, dada una clave, devolvernos el dato asociado. Esta función es parcial. No puede estar definida para todas las combinaciones de parámetros. Cuando la clave no está en la lista, no hay ningún dato que devolver; cualquier dato que devolvamos podría hacerle creer al que usa la función que la clave existe y lo tiene asociado. Cuando se le pase como argumento una clave que no esté en la lista, la función va a dar un error. Se supone que antes de llamarla un programador tiene que asegurarse de que la clave esté, puede ser con una función así:

```
esta :: clave -> [(clave,dato)] -> Bool
```

Si queremos evitarle esta pregunta a quien use nuestra función `buscar`, podemos convertirla en total usando `Maybe`:

```
buscar2 :: clave -> [(clave,dato)] -> Maybe dato
```

Cuando la clave no esté, en lugar de dar error, `buscar2` va a devolver `Nothing` y el que la use va a poder actuar en consecuencia. Por ejemplo, mostrando un error significativo por pantalla. Si `Nombre` es un tipo que nos permite identificar a un empleado y `Empleado` es el tipo `(Nombre, Int)`:

```
sueldo :: Nombre -> [Empleado] -> Int
sueldo n es = valSueldo (buscar2 n es)
valSueldo Nothing = error "No pertenece a la empresa!"
valSueldo (Just s) = s
```

La familia de tipos de `Either` también es útil y se define así:

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

Por ejemplo, los siguientes son elementos de `Either Int Bool`:

- `Left 1`
- `Left (-30)`
- `Right True`
- `Right False`

`Either` representa la unión disjunta dos conjuntos. Para distinguir de cuál de los dos tipos viene un elemento, se usa `Left` o `Right`. Sirve para poder manejar elementos de más de un tipo manteniendo el tipado fuerte. Por ejemplo, las listas deben tener todos sus elementos de un mismo tipo. Tenemos los tipos `[Int]` y `[Bool]`, pero ¿cómo hacemos para armar una lista que tenga algunos elementos enteros y otros que sean valores de verdad? Usamos el tipo `[Either Int Bool]`. Una lista de este tipo podría tener como primer elemento `Left 5` y como segundo `Right False`.

También podemos usarlo como resultado de una función para poder devolver valores de distintos tipos:

```
claveODato :: (clave,dato) -> Bool -> Either clave dato
claveODato (c,d) True = Left c
claveODato (c,d) False = Right d
```

Para terminar, se puede aplicar `Either` para distinguir la forma en que se calculó un valor:

```
mostrar :: Either Int Int -> String
mostrar (Left t) = show t ++ " Celsius"
mostrar (Right t) = show t ++ " Fahrenheit"
```

Sinónimos de tipo

Es común querer darle un nombre nuevo a un tipo de Haskell. Se hace con la cláusula `type`. No se trata de una forma de crear un nuevo tipo, simplemente crea un **sinónimo de tipo**. El nombre nuevo y el anterior pueden usarse alternativamente, son equivalentes. A veces se usa esta técnica para nombrar una instancia particular de un tipo paramétrico:

- `type String = [Char]`
- `type IntOChar = Either Int Char`

También es común renombrar tipos existentes con nombres más significativos desde el punto de vista del problema que estamos resolviendo:

- `type Nombre = String`
- `type Sueldo = Int`
- `type Empleado = (Nombre, Sueldo)`
- `type Direccion = String`
- `type Persona = (Nombre, Direccion)`

Por ejemplo, el tipo `Persona` es un par de dos `String`, pero si usáramos directamente el tipo `(String, String)`, sería muy difícil para alguien que leyera el código entender qué representa cada una de las componentes.

También se pueden crear sinónimos paramétricos, manteniendo uno o más parámetros del tipo original:

- `type Lista a = [a]`
- `type IntY a = (Int, a)`

Dado que no es una forma de definir tipos, no admite recursión (el nuevo nombre no puede aparecer a la derecha del signo `=`).

Tipos Recursivos

Cuando clasifiqué los tipos algebraicos, les mencioné los **tipos de datos recursivos**. Lo que los diferencia de las otras categorías es que el tipo que estamos definiendo aparece como argumento de alguno de sus constructores. Más adelante, voy a dar unos ejemplos más significativos. Por ahora, voy a usar un par de tipos que pueden no tener un sentido práctico para ustedes, aunque sirven para entender de qué estoy hablando:

- `data N = Z | S N`
- `data BE = TT | FF | AA BE BE | NN BE`

Veamos qué elementos tiene cada uno de estos tipos, para saber cómo son los nuevos dominios que estamos definiendo. El tipo `N` tiene, por supuesto al elemento `Z`, que es un constructor sin argumentos (como el valor `False` en `Bool`, `Calor` en `Sensacion` o `Nothing` en `Maybe a`). El otro constructor, `S`, fabrica un `N` a partir de otro `N`, agregándole una `S` al principio. Como por ahora solamente conocemos un `N` que es `Z`, podemos formar únicamente el elemento `S Z`. Este es un nuevo `N`, así que podemos usarlo con el constructor `S` para fabricar otro: `S(S Z)`, y así sucesivamente. En consecuencia, los elementos del tipo `N` son `Z`, `S Z`, `S(S Z)`, `S(S(S Z))`, `S(S(....(S Z))...)`. ¿Se les ocurre qué podríamos representar con este tipo?

En cuanto al tipo `BE`, es un poco más complejo porque tiene dos constructores recursivos, uno de ellos con dos parámetros del mismo tipo. Tenemos dos elementos que son `TT` y `FF`, constructores sin parámetros, constantes del tipo. Podemos usarlos con el constructor `NN` para formar valores parecidos a los del tipo `N` anterior: `NN(NN(...(NN TT))...)`, y también `NN(NN(...(NN FF))...)`. O podemos combinarlos con `AA`

formando los términos `AA TT FF`, `AA FF FF`. O usar los `BE` que construimos con `NN` como argumentos de `AA` así: `AA (NN(NN TT)) FF`. Esta es una forma de combinar ambos constructores recursivos. También se puede hacer la combinación inversa, aplicando `NN` a términos que comienzan con `AA`: `NN(NN(AA TT NN(TT)))`. Y, por último, usar términos que contengan `AA` como argumentos de `AA`: `AA(AA(NN FF)(TT)) (AA(AA(NN TT) TT) FF)`. Este tipo también podría representar uno más conocido, ¿se imaginan cuál?

Como habrán notado, la definición de tipos recursivos genera una explosión combinatoria de valores posibles. Sin embargo, nos da al mismo tiempo una forma de controlarla. Si bien los dominios tienen muchos términos (expresiones del tipo), todos tienen que estar creados con uno de los constructores aplicados a cero o más argumentos (recursivos o no). Usando pattern matching, podemos definir funciones recursivas sobre cualquier término mediante **recursión estructural**. Veamos algunos ejemplos:

- ```
  size :: N -> Int
  size Z = 0
  size (S x) = 1 + size x
  ```
- ```
  addN :: N -> N -> N
  addN Z m = m
  addN (S n) m = S (addN n m)
  ```
- ```
  contarAes :: BE -> Int
  contarAes FF = 0
  contarAes TT = 0
  contarAes (NN b) = contarAes b
  contarAes (AA b1 b2) = 2 + contarAes b1 + contarAes b2
  ```

Para asegurarnos de poder aplicar la función a todas las expresiones del tipo, tenemos que poner por lo menos una ecuación para cada constructor. Y para que el paso recursivo sea correcto, conviene usar, del lado derecho del igual, subexpresiones de la que aparece del izquierdo.

Listas

El tipo recursivo más usado en Haskell es el de las **listas**. Esto se debe a que el tratamiento de secuencias suele usarse en la solución de muchos problemas. Por eso el preludio y otros módulos contienen una gran cantidad de funciones para manipular listas. Como dije antes, las listas son tipos algebraicos con una sintaxis particular (por ejemplo, usando símbolos como los corchetes en lugar de palabras para los constructores). Para que puedan comprender mejor cómo trabajan las listas vamos a definir nosotros las listas como un tipo algebraico común y corriente, y después ver la correspondencia entre esta sintaxis y la que nos brinda Haskell (más cómoda, por cierto).

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

¿Qué elementos tiene el tipo `Lista a`? En primer lugar, noten que es un tipo paramétrico, así que podríamos instanciarlo en un tipo en particular para empezar a observarlo. Digamos que vamos a concentrarnos en `Lista Int`. Tenemos primero el elemento `Nil` (constructor sin parámetros). Como estamos representando secuencias, en este caso secuencias de enteros, vamos a
```

interpretar este elemento como la secuencia vacía, la que no tiene ningún elemento. De hecho la palabra "nil" es abreviatura de *nihil* que en latín significa, justamente, "nada" (no es que no signifique nada, significa "nada", que no es lo mismo). Y después podemos usar el otro constructor, `Cons` (abreviatura, convenientemente, de "construir") para agregarle un entero a una secuencia. Por ahora tenemos solamente la secuencia `Nil`, así que podemos formar la secuencia `Cons n Nil`, donde `n` es un entero cualquiera. Esto forma las listas unitarias, por ejemplo `Cons 2 Nil` es la lista que tiene solo al dos. Y después podemos seguir usando `Cons` para agregar otros enteros. Si quisiéramos formar la secuencia que tiene un dos, seguido de un tres, seguido de otro dos, seguido de un cero (cuatro elementos en total), escribiríamos lo siguiente: `Cons 2 (Cons 3 (Cons 2 (Cons 0 Nil))).`

Usemos la recursión estructural para sumar todos los elementos de una lista de enteros:

```
sumLista :: Lista Int -> Int
sumLista Nil = 0
sumLista (Cons n ns) = n + sumLista ns
```

Ya estamos listos para pasar a la sintaxis de listas que nos brinda Haskell. Acá está la traducción:

- `Nil` se escribe `[]`
- `(Cons x xs)` se escribe `(x:xs)`

Si escribimos con esta sintaxis la lista que usé como ejemplo antes, queda lo siguiente: `2 : 3 : 2 : 0 : []`. Acá no nos hizo falta usar paréntesis, porque el constructor `(:)` (sí, los dos puntos son uno de los dos constructores de listas y los dos corchetes el otro) asocian a derecha. De paso, los dos puntos suelen leerse "cons" por analogía con la otra forma de construcción que aprendimos. Los dos corchetes también pueden leerse "nil". Además, las listas son tan especiales en Haskell, que el lenguaje nos provee otra forma de escribir las listas, todavía más cómoda. La lista que escribí recién se puede escribir con esa notación de la siguiente manera: `[2,3,2,0]`. Esto no es más que una forma conveniente de escribir lo mismo que antes (como se puede ver si aplicamos una función definida por pattern matching sobre esta última).

Voy a rescribir la función que definí sobre el tipo `Lista`, ahora sobre las listas de Haskell. Y voy a agregar algunas funciones más sobre listas, para seguir mostrándoles cómo se usa la recursión estructural.

- ```
  sum :: [Int] -> Int
  sum [] = 0
  sum (n:ns) = n + sum ns
  ```
- ```
  length :: [a] -> Int
  length [] = 0
  length (x:xs) = 1 + length xs
  ```
- ```
  (++) :: [a] -> [a] -> [a]
  [] ++ ys = ys
  (x:xs) ++ ys = x : (xs ++ ys)
  ```
  ```

Todas estas funciones están definidas en el preludio. La tercera tiene una sintaxis especial que no habíamos visto. Si en lugar de ponerle a una función un nombre que empiece con una letra, usamos un símbolo, después se puede aplicar la función en forma **infija** (entre sus operandos, en vez de antes del primero) al escribir expresiones. Cuando queremos hablar de la función sin aplicarla a operandos o usarla en forma **prefija** (como las demás funciones), tenemos que ponerla entre paréntesis, como hice al declarar el tipo. Los **operadores**, estas funciones cuyos nombres son símbolos, tienen sus propias reglas. Si les interesa el tema, lean algún manual de Haskell, porque acá no vamos a profundizar mucho en estos asuntos avanzados de notación.

Las tres funciones que definí recién (por si no se dieron cuenta) sirven para sumar los elementos de una lista de enteros, contar la cantidad de elementos de una lista y concatenar (pegar) dos listas, respectivamente. Ya que definimos las listas usando dos tipos isomorfos (con la misma forma), aunque no equivalentes para Haskell (no puedo pasarle un argumento de tipo `Lista a` a una función que espera uno de tipo `[a]`, ni viceversa), podríamos definir funciones de traducción de ida y de vuelta.

- ```
  lista2list :: Lista a -> [a]
  lista2list Nil = []
  lista2list (Cons x xs) = x : lista2list xs
  ```
- ```
  list2lista :: [a] -> Lista a
  list2lista [] = Nil
  list2lista (x : xs) = Cons x (list2lista xs)
  ```

Ya está, ahora podríamos definir `sum` usando `sumLista`, por ejemplo:

```
sum ns = sumLista (list2lista ns)
```

Para terminar con las listas, voy a usarlas en la resolución de un problema para llamar la atención de ustedes sobre cómo funciona la transparencia referencial en la práctica. Esto les va a resultar un poco más extraño a quienes hayan programado en lenguajes que respondan a otros paradigmas no funcionales. A los demás tal vez les parezca normal, porque los lenguajes funcionales, a diferencia de los de algunos otros paradigmas, se parecen mucho a la notación matemática.

El enunciado del problema a resolver sería algo como “Quiero cambiar el primer elemento de una lista de naturales por un cuatro.” Y la verdad es que esto, así planteado, no se puede hacer en un lenguaje funcional. ¿Por qué? Porque en los lenguajes funcionales no hay objetos que *cambien* de valor. Una expresión siempre tiene que valer lo mismo, aparezca donde aparezca. Esta es la transparencia referencial, ¿se acuerdan? Si la lista tiene un nombre, digamos `listaUno`, definida así

```
listaUno :: [Int]
listaUno = [1, 2, 3, 4]
```

entonces `listaUno` nunca va a poder tener otro valor que no sea este. Estas ecuaciones tienen que ser ciertas en todo momento del uso y en todo lugar del programa. La visión de la transparencia referencial es que los valores, en este caso una lista, no se pueden modificar, en el mismo sentido en que el

número 2 siempre va a tener el mismo valor a pesar de que podemos aplicarle operaciones y obtener resultados a partir de él.

En consecuencia, el enunciado del problema va a tener que ser reemplazado por este otro: "Dada una lista, quiero obtener *otra* que coincida con ella en todas las posiciones, excepto en la primera, que debe ser un cuatro." Esto es matemáticamente muy razonable, si tengo dos listas que difieren en un elemento, no son la misma lista.

La función que realiza esta operación es sencilla de definir:

```
ponerCuatro :: [Int] -> [Int]
ponerCuatro (x : xs) = (4 : xs)
```

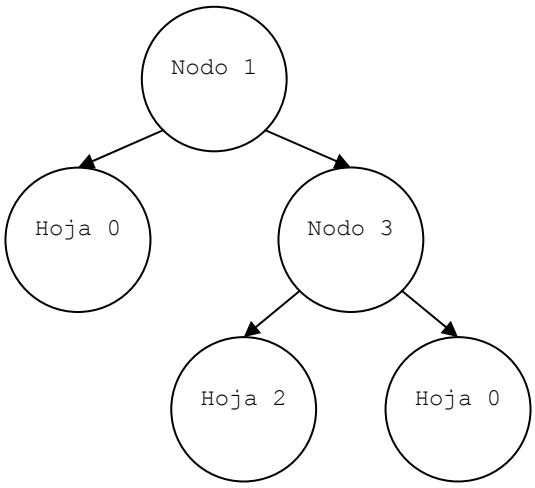
Esta función no cambia ningún elemento de la lista, crea una lista nueva parecida a la primera. De hecho tuvimos que repetir `xs` en el resultado porque si no el resto de los elementos no hubiera estado. Si se la aplicamos a `listaUno`, vamos a obtener la lista `[4, 2, 3, 4]`, que es una lista nueva. El valor de `listaUno` va a seguir siendo siempre el mismo, hagamos lo que hagamos.

Faltaría tal vez escribir una ecuación para explicar qué hacer en el caso de la lista vacía. Siguiendo la primera especificación que teníamos, que hablaba de *cambiar* un elemento por otro, es lógico dejar la definición así como está, como función parcial. Cuando alguien trate de aplicársela a la lista vacía, la función va a dar error, porque no hay ningún elemento que cambiar por un cuatro. Ahora, si nos atenemos a la nueva especificación, podríamos devolver la lista `[4]` que tiene como único elemento al número cuatro. Existe una tercera opción que consiste en devolver la lista vacía. Acá tenemos un ejemplo de por qué es importante contar con una especificación bien precisa antes de empezar a programar.

Árboles

Dejo un rato a las listas tranquilas, para pasar a contarles sobre otro tipo de datos recursivo que es habitual encontrar en el transcurso de la solución de un problema complicado: los árboles.

Para simplificar el tratamiento, vamos a trabajar con árboles binarios, con datos en los nodos internos y en las hojas:



Llamemos a este árbol `aej` y definámoslo, junto con el tipo `Arbol`.

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

aej :: Arbol Int
aej = Nodo 1  (Hoja 0)
              (Nodo 3 (Hoja 2) (Hoja 0))
```

Ahora voy a definir un par de funciones para mostrarles cómo usar la recursión estructural sobre árboles:

- ```
  hojas :: Arbol a -> Int
  hojas (Hoja x) = 1
  hojas (Nodo x i d) = hojas i + hojas d
  ```
- ```
  altura :: Arbol a -> Int
  altura (Hoja x) = 0
  altura (Nodo x i d) = 1 + (altura i `max` altura d)
  ```

La primera cuenta el total de hojas que tiene un árbol y la segunda mide su altura (máxima distancia entre una hoja y la raíz). En esta usé de nuevo una notación que no habíamos visto. La función `max` calcula el máximo entre dos números. Según habíamos visto hasta ahora, se usaría con la notación prefija, `max n1 n2`. Las comillas invertidas sirven para "convertir" funciones binarias en operadores infijos y poder usarlas con esta notación que, a veces, es más cómoda. En este caso particular nos ahorró encerrar los dos términos entre paréntesis, porque los operadores tienen una precedencia muy baja.

Ahora les voy a mostrar una función sobre árboles que trabaja en forma parecida a la función `ponerCuatro` que vimos para las listas. Se podría decir que *cambia* los números 2 que haya en las hojas por números 3. Aunque sabemos que, en realidad, fabrica árboles nuevos, con los mismos datos que el recibido como argumento, excepto algunos.

- ```
  cambiar2 :: Arbol Int -> Arbol Int
  cambiar2 (Hoja n) | n==2 = Hoja 3
  cambiar2 (Hoja n) | otherwise = Hoja n
  cambiar2 (Nodo n i d) = Nodo n (cambiar2 i) (cambiar2 d)
  ```

La palabra `otherwise` significa "en cualquier caso" y representa al valor de verdad `True`.

Como ejercicio, pueden probar reducir `cambiar2 aej`, para ver cómo se comporta la función. Voy a darles un par de ejemplos más de funciones sobre árboles, para que hagan más pruebas:

- ```
  duplA :: Arbol Int -> Arbol Int
  duplA (Hoja n) = Hoja (n*2)
  duplA (Nodo n i d) = Nodo (n*2) (duplA i) (duplA d)
  ```
- ```
  sumA :: Arbol Int -> Int
  sumA (Hoja n) = n
  sumA (Nodo n i d) = n + sumA i + sumA d
  ```

El ejercicio es reducir `duplA aej` y `sumA aej`.
  ```

Estas funciones van “recorriendo” un árbol y operando con cada nodo. Podrían hacerlo en cualquier orden, dando el mismo resultado. Pero a veces es importante en qué orden recorremos un árbol para aplicarle alguna operación. Por ejemplo, si buscáramos en qué ubicación está la primera aparición de un dato en un árbol, la interpretación de “primera” puede cambiar de acuerdo al orden en que recorramos nodos y hojas. Les voy a presentar dos funciones que arman una lista con los elementos de un árbol. Ambas respetan el orden que tienen los elementos en el árbol, aunque el criterio para respetar ese orden es distinto.

- ```
  inOrder, preOrder :: Arbol a -> [a]
  inOrder (Hoja x) = [x]
  inOrder (Nodo x i d) = inOrder i ++ [x] ++ inOrder d
  ```
- ```
  preOrder (Hoja x) = [x]
  preOrder (Nodo x i d) = x : (preOrder i ++ preOrder d)
  ```

Como hicieron (supuestamente) con las funciones anteriores, prueben reducir `inOrder aej` y `preOrder aej`, y comparen las listas resultantes. Hay un tercer orden posible que se llama `posOrder`. ¿Se imaginan en qué consiste? ¿Se animan a definir la función correspondiente?

Una vez que convertimos el árbol en lista de acuerdo al orden que necesitamos, podemos aprovechar las funciones que ya tenemos definidas sobre listas para realizar operaciones sobre los elementos del árbol. Por ejemplo, podríamos haber definido `sumA` así:

```
sumA a = sum (preOrder a)
```

La mayoría de las funciones que estuvimos definiendo hasta ahora son polimórficas, ¿se acuerdan de que les había prometido que íbamos a aplicar el polimorfismo a casos más interesantes que el de la identidad? Bueno, escribimos y usamos un montón de funciones (como `preOrder`) que se basan solamente en la estructura de sus argumentos, sin importar su valor, y son aplicables a muchos tipos de datos existentes y por existir.

Expresiones aritméticas

Voy a terminar con otro tipo recursivo, de estructura muy similar a la de los árboles. Es un tipo cuyos elementos son expresiones aritméticas como `4*4` o `(2*3)+4`. O sea que voy a definir un tipo cada uno de cuyos elementos represente una de estas expresiones, que pueden ser constantes numéricas, o sumas o productos de otras dos expresiones. La oración que acabo de escribir me indica bastante claramente cómo tengo que definir el tipo:

```
data ExpA = Cte Int | Suma ExpA ExpA | Mult ExpA ExpA
```

Esto se lee casi textualmente como la anteúltima oración del párrafo anterior. Según esta definición, la expresión `2` se representa como `Cte 2`; la expresión `4*4` se representa como `Mult (Cte 4) (Cte 4)` y la expresión `(2*3)+4` se representa como `Suma (Mult (Cte 2) (Cte 3)) (Cte 4)`.

Ya que solamente estamos considerando expresiones aritméticas sin variables, siempre vamos a poder evaluarlas (reducirlas) hasta obtener un valor numérico. Definamos una función para hacerlo.

- ```
  evalEA :: ExpA -> Int
  evalEA (Cte n) = n
  evalEA (Suma e1 e2) = evalEA e1 + evalEA e2
  evalEA (Mult e1 e2) = evalEA e1 * evalEA e2
  ```

¿Muy simple no? Y, como quien no quiere la cosa, programamos una calculadorita. Para probarla, reduzcan las siguientes expresiones:

- `evalEA (Suma (Mult (Cte 2) (Cte 3)) (Cte 4))`
- `evalEA (Mult (Cte 2) (Suma (Cte 3) (Cte 4)))`

También pueden traducirlas a la notación habitual y ver si el resultado que redujeron es el correcto.

Tipos abstractos

Motivación

Ya vimos cómo crear y cómo usar tipos de datos algebraicos. Pero les dije que, a la hora de usar un tipo, se lo podía recibir también como un tipo abstracto. Quiere decir que muchas veces vamos a recibir (o a entregar) un tipo de datos sin permitir el acceso a su representación. Para eso, ni siquiera se requiere que haya más de un programador involucrado. Puedo crear un tipo de datos (como un tipo algebraico) que voy a usar yo mismo y supuestamente nadie más, pero ocultar su implementación de forma tal que en otras partes del programa me esté prohibido accederla.

¿Por qué? Por dos motivos. Primero porque la representación del tipo puede no cumplir algunos requisitos básicos, como los que pusimos para crear tipos de datos algebraicos. Cuando definimos el tipo de los racionales en forma algebraica a partir de dos enteros, nos dimos cuenta de que no estábamos cumpliendo con estas pautas. Les recuerdo:

```
data Racional = R Int Int
```

Podemos usar esta representación, pero tendríamos que tener muchísimo cuidado para nunca construir un racional cuyo segundo componente fuera cero. Y cada vez que definiéramos una nueva función, tendríamos que asegurarnos de que se comportara igual para dos formas distintas de representar el mismo racional. Una función `numerador` que devuelva la primera componente no estará bien definida, porque dará resultados distintos para `R (-1) (-2)` y para `R 2 4`, que son el mismo número (recuerden que no estamos representando las fracciones, sino los números racionales).

Como dije, tendríamos que tener muchísimo cuidado al usar este tipo de datos. Y lo ideal es que el lenguaje nos ayude a que nos cuidemos. Tiene que proveernos recursos sintácticos para que podamos avisarle que no queremos que se pueda usar esta representación interna desde más que unas poquitas funciones, y mostrarnos un mensaje de error si intentamos violar este acuerdo.

El segundo motivo por el que podemos querer prohibir el acceso a la representación interna es para que el programa sea más barato de mantener. Si el acceso por pattern matching está limitado solamente a algunas funciones y en el futuro decidimos cambiar esa representación interna, el cambio que tengamos que hacer se va a limitar también a esas funciones. En lo posible van
```

a aparecer muy cerca una de otra en el código (en el mismo archivo). Si no le pidiéramos al lenguaje esta protección, correríamos el riesgo de usar (nosotros mismos u otro programador) la implementación del tipo en muchos otros programas. Cuando la reemplacemos por otra, van a dejar de funcionar todos esos programas hasta que los modifiquemos uno por uno.

Espero haberlos convencido de que es bueno que el lenguaje nos brinde alguna manera de abstraernos de la representación interna de los tipos y de que vale la pena aplicar estos mecanismos. A continuación, les voy a contar cómo usar los tipos abstractos que recibimos ya implementados y después cómo hacer para presentar como abstractos los tipos que creamos nosotros.

Uso de tipos abstractos

Cuando recibimos un **tipo abstracto de datos**, lo que se nos da es el **nombre del tipo**, los **nombres de sus operaciones** básicas, los **tipos de las operaciones** y una **especificación** del funcionamiento de esas operaciones. Como siempre, la especificación puede ser más o menos formal y, por lo menos en el caso de Haskell, no va a ser chequeada por el lenguaje.

¿Cómo hacemos para utilizar el tipo? A través de sus operaciones y únicamente así. No solo por convicción, porque no tenemos otro remedio.

Si usamos un tipo de datos abstracto que representa a los racionales, no nos importa cómo fue definido. Quizás lo hicieron con un tipo algebraico como propuse yo antes, pero de todos modos no podemos aplicar pattern matching. Nos dan una serie de operaciones que podrían ser las siguientes:

- `crearR :: Int -> Int -> Racional`
- `numerR :: Racional -> Int`
- `denomR :: Racional -> Int`

Y la especificación nos dice, por ejemplo, que el numerador y el denominador corresponden siempre a alguna forma normalizada (típicamente, la máxima simplificación) con el signo en el numerador. No sabemos cuándo se produce la simplificación para normalizar la representación, podría ser al construir el número en `crearR` o al evaluarlo, en `numerR` y `denomR`. Esto no nos preocupa, en tanto se cumpla la especificación. Si es así, pasa que

```
numerR (crearR (-1) (-2)) == numerR (crearR 2 4)
```

La especificación también aclara que `crearR n 0` va a producir un error.

Tal vez nos den además algunas operaciones básicas, como la suma y la multiplicación de racionales, pero también podríamos definirlas nosotros:

- ```
  sumaR, multR, divR :: Racional -> Racional -> Racional
  r1 `sumaR` r2 = crearR
       (denomR r2 * numerR r1 + denomR r1 * numerR r2)
       (denomR r1 * denomR r2)
  ```
- ```
  r1 `multR` r2 = crearR
       (numerR r1 * numerR r2)
       (denomR r1 * denomR r2)
  ```
- ```
  r1 `divR` r2 = crearR
  ```
  ```

```
    (denomR r2 * numerR r1)
    (denomR r1 * numerR r2)
```

Cuando se trabaja con tipos abstractos de datos, normalmente hay que identificar tres **roles** para las personas involucradas: diseñador, implementador y programador-usuario. Puede ser que una persona ocupe más de uno de ellos simultáneamente, pero en cada momento debería tener claro en cuál de ellos está. También puede ser que alguno de los roles lo lleve a cabo un equipo de trabajo (sobre todo, el último).

El **diseñador** es quien establece qué operaciones proveerá el tipo y escribe las especificaciones correspondientes. También puede decidir o sugerirle al implementador qué representación interna usar. La decisión podría basarse en qué operaciones serán las más frecuentes y, por lo tanto, deben calcularse más rápido.

El **implementador** se encarga de escribir el código necesario para definir el tipo abstracto, su representación y las funciones básicas identificadas por el diseñador. Es importante marcar la diferencia entre QUÉ hacer (decisiones tomadas por el diseñador) y CÓMO hacerlo (código escrito por el implementador para cada función).

El **programador-usuario** se limita a utilizar las operaciones, sin saber cómo están implementadas. O sabiéndolo, pero no pudiendo usar este conocimiento en sus programas. Además de las ventajas que vimos, está la de poder desentenderse de la implementación, olvidarse de qué estructuras son las que uno está manipulando realmente y usar el tipo como si fuera primitivo del lenguaje. Algunos tipos que venimos usando, como `Int` o `Float` son tipos abstractos: no sabemos (y preferimos no saber) cómo están representados internamente. Mientras podamos usarlos correctamente en operaciones aritméticas, no nos importa de qué estén hechos sus elementos por dentro.

Quiero darles un par de ejemplos más de tipos abstractos, que es un tema muy importante. El primero es el `Diccionario`. Voy a usar un diccionario que guarda definiciones para ciertas palabras (como un diccionario de la lengua). Pero este tipo de datos, visto en forma genérica (como paramétrico), se suele usar para representar cualquier función discreta.

Desde el punto de vista del implementador, probablemente, el diccionario se guarde fuera de la memoria de la máquina en algún medio externo que tenga más capacidad y tolere mejor las fallas del sistema. Por ahora no vamos a aprender cómo comunicarnos con dispositivos de este tipo desde Haskell. De todas maneras, esto no debería interesarnos como programadores-usuarios del tipo abstracto. Las operaciones que nos brinda el diseñador son estas:

- `buscar :: Diccionario -> Palabra -> Maybe Definicion`
- ```
  agregar :: (Palabra, Definicion) ->
             Diccionario -> Diccionario
  ```
- `eliminar :: Palabra -> Diccionario -> Diccionario`
- `castellano :: Diccionario`

Piensen un rato en la segunda y en la tercera desde el punto de vista de la transparencia referencial. Estas funciones no modifican, no *cambian* un
  ```

diccionario. Lo que hacen es *crear* un diccionario nuevo a partir de uno existente y unos datos más. Por ejemplo, la expresión `agregar ("culebrón", "telenovela") castellano` va a construir un diccionario nuevo al que yo podría preguntarle qué quiere decir "culebrón". Pero la expresión `castellano` va a seguir devolviendo el mismo diccionario que antes. ¿Cómo se llama el nuevo diccionario? No tiene un nombre, la forma de hablar de él es mediante la expresión entera que escribí más arriba. Entonces, si quisiéramos hacer la pregunta, escribiríamos
`buscar (agregar ("culebrón", "telenovela") castellano) "culebrón"`
y esto nos devolvería `Just "telenovela"` (el resultado de `buscar` es de tipo `Maybe Definicion`.)

Además de las operaciones y sus tipos, el diseñador nos entrega una especificación que define su comportamiento. La especificación puede ser informal, simplemente explicando qué se espera de cada operación, lo cual puede inducir a confusiones. Es mejor utilizar un lenguaje formal como el de la lógica de primer orden. En ese caso, vamos a recibir un conjunto de predicados, uno de los cuales podría ser
$(\forall p)\ (\forall def).(\forall dict)$ agregar (p,def) (agregar (p,def) dict)= agregar (p,def) dict.

O sea: agregar dos veces una palabra a un diccionario es lo mismo que agregarla una vez. Otro predicado podría indicar que no importa en qué orden se agreguen dos palabras a un diccionario. Otro, que si se saca una palabra su definición desaparecerá.

Veamos el último ejemplo de uso de un tipo abstracto de datos: los conjuntos. La teoría de conjuntos aportó herramientas muy poderosas al razonamiento matemático. La gran mayoría de los conceptos de la matemática tal como la conocemos hoy está definida usando conjuntos. A pesar de esto, como ya vimos, el tipo de datos preferido para representar colecciones en programación funcional no es el de los conjuntos sino el de las listas. ¿Por qué pasa esto? Fundamentalmente, porque los conjuntos no pueden representarse con un tipo algebraico que cumpla las condiciones que pusimos. Piensen un rato en alguna definición que se les ocurra y van a ver que los constructores siempre les permiten armar de más de una forma el mismo conjunto. (o expresiones que no representan conjuntos, según la definición).

¿Cuál es la solución? Crear un tipo abstracto para los conjuntos. Internamente, puede estar implementado con cualquiera de las estructuras que hayan pensado (si es que las pensaron). Pero las operaciones disponibles para un programador-usuario van a limitar su uso como pasó con los racionales. El tipo abstracto para los conjuntos de enteros podría ser así:

- `empty :: IntSet`
  El conjunto vacío.
- `isEmpty :: IntSet -> Bool`
  Determina si el conjunto dado es vacío.
- `belongs :: Int -> IntSet -> Bool`
  Determina si un elemento pertenece o no al conjunto.
- `insert :: Int -> IntSet -> IntSet`
  Agrega un elemento al conjunto, si no estaba. Si estaba, lo deja igual.

- `choose :: IntSet -> (Int, IntSet)`
  Elige el menor número y lo quita del conjunto.

Esta vez, la especificación la di informalmente, usando la "metáfora" de la modificación. En realidad, acuérdense de que las operaciones no modifican los conjuntos, construyen conjuntos nuevos. Pero es fácil explicar las cosas en estos términos. La última operación no *quita* nada del conjunto, que sigue igual que antes; lo que hace es devolver un entero y otro conjunto, con todos los elementos del primero menos ese. No queda claro qué pasa con esta operación si el conjunto es vacío, se puede suponer que da error.

Como programadores-usuarios, definamos una operación nueva: la unión.

```
union :: IntSet -> IntSet -> IntSet
union p q | isEmpty p = q
union p q | otherwise = unionAux (choose p) q
unionAux (x,p') q = insert x (union p' q)
```

Observen que esto pudimos hacerlo sin tener idea de cuál es la representación de los conjuntos, usando solamente las operaciones provistas. Si cambiara la representación, el implementador solamente tendría que rescribir las ecuaciones para las funciones propias del tipo abstracto (`empty`, `isEmpty`, `belongs`, `insert`, `choose`). La función `union` y cualquier otra que definamos en este u otros programas van a quedar intactas. El ejemplo también muestra que la recursión no es privativa del pattern matching. La función `union` está definida en forma recursiva (a través de `unionAux`), pero no usa recursión estructural ¡porque no conocemos la estructura!

Definición de tipos abstractos

Paso por un rato al rol de implementador, así les muestro cómo se escribe un tipo abstracto que puedan usar otros programadores. En Haskell se hace mediante **módulos**. Son archivos de texto que contienen una parte de un programa. Por ejemplo, uno o más tipos de datos con sus funciones. O, simplemente, un grupo de funciones (preferentemente, de utilidad relacionada).

Los módulos le aportan a Haskell la posibilidad de realizar encapsulamiento.

El **encapsulamiento** consiste en agrupar una estructura de datos con las funciones básicas para trabajar con ella. Usando módulos, un programa en Haskell, en lugar de ser una lista larga de definiciones y tipos, va a estar formado por varias "cápsulas" que son los módulos. Cada una contiene un (o tal vez más) tipo de datos junto con sus funciones específicas y oculta parte de su contenido.

Cuando dije **oculta** me refería a que cuando escribo un módulo tengo que indicar cuáles de los nombres que definí en él **exporta** (cuáles van a poder usarse desde afuera y cuáles no). Una aplicación del ocultamiento es la de crear funciones auxiliares (como la que llamé `unionAux`). No queremos que nadie la use en otro lado, ni impedirles a otros programadores usar ese nombre para sus funciones, entonces no la exportaríamos. El ocultamiento también es una herramienta ideal a la hora de crear tipos de datos abstractos, porque justamente se trata de ocultar la representación interna (y a veces otros detalles de implementación).

Ya hablé mucho en el aire, pasemos a un ejemplo. Primero veamos uno de módulo. No voy a implementar un tipo abstracto todavía, sino a agrupar la funcionalidad del tipo de los complejos en un módulo. Como se representan muy bien mediante un tipo algebraico, vamos a exportar el tipo entero.

```
module Complejos
    (Complejo(..), parteReal, parteIm) where

data Complejo = C Float Float

parteReal, parteIm:: Complejo -> Float
parteReal(C r i) = r
parteIm (C r i) = i
```

La palabra `module` introduce el módulo, poniéndole nombre e indicando qué es lo que exporta. Al escribir el nombre del tipo seguido de dos puntos suspensivos entre paréntesis, se exporta el nombre del tipo y sus constructores. Quiere decir que cualquier programador que use este módulo va a poder hacer pattern matching sobre elementos del tipo `Complejo`. Después de la lista de nombres exportados viene el `where` y las definiciones correspondientes. Si no se pone la lista de exportación (`Module Complejos where...`), se exportan TODOS los nombres definidos. Si se pone una lista vacía (`Module Complejos () where...`), ¡no se exporta NADA!

Ya vieron la sintaxis de los módulos, ahora pasemos a un tipo abstracto. El de los racionales, como vimos, es un buen candidato:

```
module Racionales
    (Racional, crearR, numerR, denomR) where

data Racional = R Int Int

crearR :: Int -> Int -> Racional
crearR n d = reduce (n*signum d) (abs d)

reduce :: Int -> Int -> Racional
reduce x 0 = error "Racional con denom. 0"
reduce x y = R (x `quot` d) (y `quot` d)
  where d = gcd x y

numerR, denomR :: Racional -> Int
numerR (R n d) = n
denomR (R n d) = d
```

Las funciones `signum` (signo), `abs` (valor absoluto), `quot` (división entera) y `gcd` (*gratest common divisor*, máximo común divisor) están definidas en el preludio. En la lista de exportación del módulo no puse los dos puntos suspensivos entre paréntesis después de la palabra `Racional`. Por lo tanto, estoy exportando solamente el nombre del tipo y NO sus constructores. Esto convierte el tipo en abstracto para los programadores que lo usen. En la lista tampoco aparece `reduce`, que es una función auxiliar para el tipo.

¿Cómo hace el programador-usuario para indicar en su programa (en otro módulo) que quiere incorporar este tipo de datos? Tiene que usar la cláusula `import`. Puede importar todos los nombres exportados por un módulo

indicando sólo el nombre del módulo, o importar solamente algunos de ellos aclarando entre paréntesis cuáles.

```
module Main where

import Complejos
import Racionales (Racional, crearR)

miPar :: (Complejo, Racional)
miPar = (C 1 0, crearR 4 2)
```

Si en el módulo `Main` pusiera cualquiera de las siguientes expresiones, recibiría mensajes de error al tratar de cargarlo por estar intentando usar nombres ocultos:

- `numerR (snd miPar)`
- `reduce (crearR 4 2)`
- `R 4 2 + R 2 1`

El último ejemplo que voy a darles es el de los conjuntos de enteros. Voy a aprovecharlo para enseñarles una forma más de crear de tipos, que es común al implementar tipos de datos abstractos.

Ya vimos cómo crear sinónimos de tipos. Cuando creamos un sinónimo para un tipo existente, contamos con un nombre más para referirnos al mismo tipo. El tipo original y el sinónimo son intercambiables, puedo declarar que una función recibe un parámetro de un tipo y después pasarle un valor que fue declarado como perteneciente a su sinónimo.

Pero otras veces, queremos crear un tipo *nuevo* que se represente de la misma manera que uno existente. Destaqué la palabra “nuevo” porque la idea en estos casos es que los dos tipos NO puedan intercambiarse. Esta notación existe porque, si la representación que elegimos para un tipo abstracto es un tipo ya existente, no nos conviene usar un sinónimo de tipo. Si lo hiciéramos, correríamos el riesgo de que un programador-usuario aprovechara la sinonimia en su programa con los consiguientes perjuicios: podría acceder a la representación tomando elementos conceptualmente iguales como diferentes o creando valores inválidos para el nuevo tipo; y además dejaría el programa atado a esta representación.

La manera de definir un nuevo tipo con la misma representación que uno existente es usando la cláusula `newtype`, cuyo nombre indica muy bien su aplicación. En el caso de los conjuntos de enteros, podríamos representarlos *internamente* con listas de enteros. Por supuesto, vamos a tener que encerrarlos en un tipo abstracto para que nadie pueda, por ejemplo, distinguir entre las listas `[1, 3, 2, 4, 3]` y `[1, 2, 3, 4]` que deberían representar al mismo conjunto, por tener los mismos elementos.

```
newtype IntSet = Set [Int]
```

Esta cláusula es muy similar a una cláusula `data`: define el nuevo tipo de datos y su constructor (que luego ocultaremos). ¿Por qué no escribí `data`? Las diferencias son pocas, pero fundamentales. La más importante es llamarle la atención sobre el hecho de que se trata solamente de un renombre a quien lea la implementación en el futuro (otro implementador encargado de modificarla,

alguien que tenga que analizarla para ver qué hace o demostrar que cumple la especificación, el mismo implementador dentro de un tiempo, etc.). Por esto mismo `newtype` admite un solo constructor con un único parámetro: así queda claro que no estamos creando nuevos elementos, simplemente renombrando elementos existentes (con un nombre no reemplazable por el anterior).

Otra diferencia entre `newtype` y `data` tiene que ver con el rendimiento de un programa. Los intérpretes de Haskell representan internamente los valores de tipos algebraicos mediante estructuras más difíciles de acceder que las de los tipos contenidos en ellas. En cambio, los elementos de tipos definidos con `newtype` son representados por valores del tipo original; por lo tanto, se acceden más rápidamente.

Pasemos a la implementación completa de los conjuntos de enteros. Voy a representarlos por listas ordenadas sin elementos repetidos. De esta forma, dos de estas listas van a ser iguales si y solo si representan el mismo conjunto.

```
module ConjuntoInt
    (IntSet, empty, isEmpty, belongs, insert, choose) where

import qualified List (insert)

newtype IntSet = Set [Int]

empty :: IntSet
empty = Set []

isEmpty :: IntSet -> Bool
isEmpty (Set xs) = null xs

belongs :: Int -> IntSet -> Bool
belongs x (Set xs) = x `elem` xs

insert :: Int -> IntSet -> IntSet
insert x (Set xs) | x `elem` xs = Set xs
insert x (Set xs) | otherwise = Set (List.insert x xs)

choose :: IntSet -> (Int, IntSet)
choose (Set (x:xs)) = (x, Set xs)
```

Tuve que hacer un pequeño truquito que les voy a enseñar (soy mejor profesor que mago). Hay una función `insert` que está definida en el modulo `List`. Me vino muy bien para implementar los conjuntos, porque hace exactamente lo que yo necesito: inserta un elemento en su lugar correspondiente de una lista ordenada (es una forma de decir, recuerden la transparencia referencial). El problema con el que me encontré para usarla es que yo estaba definiendo una función `insert` para conjuntos, así que no iba a poder tener en un mismo programa las dos funciones conviviendo. Lo resolví usando la palabra `qualified` al importar el nombre `insert` del módulo `List`. Esto quiere decir que para referirme a ese nombre tengo que **calificarlo** con el del módulo, o sea, tengo que escribir `List.insert`. Esto me permitió distinguir los dos `insert`.

Las funciones `null` (ver si una lista es vacía) y `elem` (ver si un elemento pertenece a una lista) están definidas en el preludio.

Con esto terminamos el tema de definición de tipos en Haskell. Dejé afuera algunos chiches de notación, como el definir campos con nombre para los tipos algebraicos. También excluí un tema conceptualmente complejo que son las clases. Esta extensión sobre el sistema de tipos de Hindley-Milner va más allá del polimorfismo al permitir definir funciones con el mismo nombre pero distinto comportamiento para determinados grupos de tipos (sobrecarga).

Computación

Para terminar con la introducción al paradigma funcional. Les voy a hablar de la computación, el cálculo del valor de las expresiones. No se trata únicamente de entender cómo hace la implementación del lenguaje para realizar el cálculo. También tiene que ver con la **semántica**, con el significado de los programas. Me refiero a que distintos modelos de computación pueden hacer que una misma expresión denote distintos valores. Entonces, cuando escribimos un programa tenemos que tener en mente cuál es el mecanismo que se va a usar para ejecutarlo, porque de ese mecanismo depende su funcionamiento.

Por suerte, no son todas las expresiones las que varían su significado de acuerdo con la forma de calcularlas. De lo contrario, podría volverse muy complicado escribir programas funcionales, lo cual, como habrán visto (¡ojalá!), no es así.

Reducción

Les hablé antes de las reducciones. Recuerden que una **reducción** consiste en reemplazar una subexpresión por otra. La subexpresión reemplazada tiene que ser una instancia del lado izquierdo de una ecuación orientada. La que la reemplaza es el lado derecho de la ecuación, instanciado de manera acorde. La **instanciación** de la que vengo hablando es la asignación de expresiones a variables de un pattern.

Por ejemplo, supongamos que tenemos lo siguiente:

- la expresión `sumar (restar 2 (amigos Juan)) 4`,
- la ecuación `restar x y = x - y`.

Para hacer una reducción primero tenemos que darnos cuenta de que la subexpresión `restar 2 (amigos juan)` es una instancia del lado izquierdo de la ecuación (con la asignación `x` ← `2`, `y` ← `(amigos Juan)`). Quiere decir que podemos reemplazarla por el lado derecho (`x - y`) instanciado según la misma asignación (`2 - (amigos Juan)`). Al reducir la expresión obtenemos `sumar (2 - (amigos Juan)) 4`.

La forma de calcular el valor de una expresión es seguir reduciendo hasta que no haya ninguna subexpresión que nos sirva, ninguna que pueda ser instancia del lado izquierdo de una ecuación. Cuando se estudia o se piensa sobre este tema, este concepto aparece muchas veces: “subexpresión que es una instancia del lado izquierdo de una ecuación”. Para un concepto que se repite bastante, este nombre es un poco incómodo, se harían engorrosos los razonamientos si uno tuviera que decir a cada rato “subexpresión que es una instancia del lado izquierdo de esta ecuación”, “subexpresión que es una instancia del lado izquierdo de esta otra ecuación”. Por eso, a las

subexpresiones que son instancias del lado izquierdo de ecuaciones, se les puso un nombre cortito: **redex**. Esta palabra es una apócope de *reducible expression* (expresión reducible) y significa "subexpresión que es una instancia del lado izquierdo de una ecuación" (aunque esto ya se lo habían imaginado).

Les había dicho que calcular una expresión era llevarla a una **forma normal** que es algo muy parecido a un valor. Ahora podemos dar una definición más exacta: una forma normal es una expresión constituida solamente por constantes y constructores. Por lo tanto, una forma normal no puede contener redexes. Por ejemplo,

- `2`
- `Just 'a'`
- `(3, True, C 5.1 (-6.2))`

son formas normales; y

- `1 + 1`
- `primerCar "abracadabra"`
- `(9 `quot` 3, 3>=2, sumarC (C 1.05 (-12.4)) (C 4.05 (6.2)))`

no son.

Con estas definiciones, el **mecanismo de reducción** se puede describir así:

1. Si la expresión está en forma normal, terminamos.
2. Si no, buscar un redex.
3. Reemplazarlo.
4. Volver a empezar.

Todos los mecanismos de reducción comparten este algoritmo. Las estrategias dependen, básicamente, del paso 2. Los distintos mecanismos difieren en la forma de buscar un redex y es lo que puede hacer que se llegue o no a una forma normal.

El objetivo de la reducción es la **normalización**: encontrar la forma normal de una expresión, si es que la tiene. La primera pregunta que se plantea es si toda expresión tiene una forma normal. La respuesta es "No". Y la segunda, dado que hay más de una estrategia para reducir una expresión, es si, de existir, la forma normal es única. La respuesta es "Sí". Esta propiedad de la reducción se llama **confluencia**: todos los caminos conducen a Roma: partiendo de una expresión, puede ser que una estrategia de reducción dé una forma normal y otra no; pero no puede ser que den dos formas normales distintas.

Como dije, los mecanismos de los que estuve hablando difieren en la búsqueda del próximo redex a reducir. Lo que varía de uno a otro es en qué orden se van tomando los redexes, por eso se llaman **órdenes de reducción**.

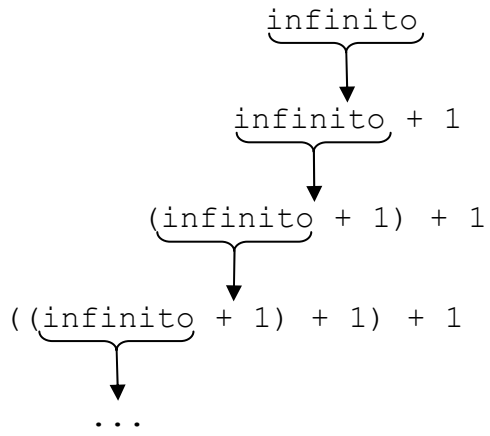
Concentrémonos en la primera de las preguntas que mencioné. Para ver que no todas las expresiones tienen forma normal, partamos de dos ecuaciones:

- ```
  infinito :: Int
  infinito = infinito + 1
  ```
- ```
  recip :: Float -> Float
  recip x | x/=0 = 1/x
  ```

Y tratemos de normalizar estas dos expresiones:

- `infinito`
- `recip 0`

Aplicando la reducción (sin importar qué orden usemos porque no hay muchos redexes de dónde elegir en estas expresiones) no vamos a llegar a formas normales en ninguno de los dos casos. En el primero, podemos seguir reduciendo todo el tiempo que queramos:



En el segundo caso, no tenemos ninguna reducción que hacer, pero la expresión no es una forma normal, porque `recip` no es una constante ni un constructor. Ya encontramos dos expresiones a las que, por lógica, no debería corresponderles ninguna forma normal. Pero yo había dicho que a toda expresión bien formada (y estas lo son, no tienen errores de sintaxis ni de tipos) tenía que corresponder un valor. Veamos cómo se resuelve la aparente contradicción.

En la visión operacional, esto no constituye un problema, porque simplemente podemos clasificar las expresiones que encontramos recién en dos grupos:

- expresiones que no terminan nunca de evaluarse
- expresiones que no pueden seguir evaluándose

En cambio, según la visión denotacional, nuestro deber es encontrar un valor para estas expresiones. Para eso se define un valor nuevo. Se llama **bottom** (que quiere decir fondo, extremo inferior) y se simboliza así: ⊥ (con una letra “T” dada vuelta).

Bottom

Bottom es un valor especial para representar en la visión denotacional las computaciones que no terminan (bien porque podrían seguir para siempre o porque llega un punto a partir del cual no hay cómo seguirlas). Es un valor teórico, no podemos incorporarlo en la visión operacional porque funciona como una especie de agujero negro: solamente por preguntar si algo es ⊥, obtenemos ⊥. Se debe a que, para tratar de descubrir si una expresión vale ⊥, tenemos que evaluarla, y entonces...

Es lo mismo decir que una expresión no tiene forma normal o que vale bottom. Estas expresiones se llaman también **indefinidas** y las que sí tienen forma normal se llaman **definidas**. Las expresiones indefinidas que puse como ejemplo eran una de tipo `Int` y la otra de tipo `Float`. Lo cierto es que se

pueden escribir expresiones de cualquier tipo que valgan ⊥. Así que todos los tipos tienen que contener el valor ⊥.

Para acercarnos a este concepto en la visión operacional, podemos definir el nombre `bottom` con ecuaciones, de forma tal que su valor sea efectivamente ⊥ y pertenezca a todos los tipos:

```
bottom :: a
bottom = bottom
```

A partir de esta ecuación, queda claro que cualquier intento de evaluar `bottom` en un programa va a hacer que la expresión en la cual aparezca valga también ⊥. Veamos una definición de función que incluya esta expresión:

```
f :: Int -> Int
f x = if x == bottom then 1 else 0
```

Digamos que ahora tratamos de evaluar la expresión `f 2`. El intérprete se va a tomar su tiempo tratando de reducir el redex `bottom` para poder llegar a un valor que pueda comparar con el `2`. Tal vez la implementación vaya usando cada vez más memoria y finalmente tenga que detenerse diciéndonos que no tiene memoria suficiente. O tal vez no termine nunca y tengamos que detener el proceso nosotros a mano. Lo cierto es que la expresión vale ⊥ por el solo hecho de haber intentado comparar algo con `bottom`.

En Haskell, tenemos una alternativa que es la función `error`.

```
error :: String -> a
```

Es una función predefinida que no tiene ninguna ecuación asociada. Cuando se intenta evaluar la expresión `error "Mensaje"`, se detiene el cálculo y se muestra `"Mensaje"` por pantalla. Si bien es una forma más de devolver ⊥, deja bien claro qué fue lo que causó el problema (siempre y cuando el mensaje en sí sea claro).

Aplicación de funciones

Como les mostré con `f`, una función puede recibir un valor definido y, sin embargo, devolver ⊥. Ya hablamos de estas funciones cuyo dominio no tiene todos los valores del tipo al cual se aplica. Se llaman **funciones parciales**. Las demás funciones, las que están definidas para todos los valores del tipo de origen, se llaman **funciones totales**. En realidad, cuando digo “todos los valores” me refiero a todos los valores definidos, todos menos ⊥.

¿Pero qué pasa cuando le pasamos ⊥ como parámetro a una función? ¿Devuelve también ⊥? No siempre. Porque, de acuerdo a las ecuaciones que la definan y al orden de reducción, podría ser que la función no precisara calcular el valor de ese parámetro para darnos un resultado (enseguida les doy ejemplos). La clasificación de funciones en parciales y totales depende de qué hace la función cuando recibe un parámetro definido. Hay otra clasificación que se basa en el resultado de la función cuando uno de sus parámetros es indefinido. Si la función devuelve también ⊥, se dice que la función es **estricta** y si devuelve un valor definido, se llama **no estricta**.

Estas dos clasificaciones son independientes, en el sentido de que pueden darse todas las combinaciones: puede haber funciones parciales estrictas y no estrictas, y funciones totales estrictas y no estrictas. Les hago un cuadrito:

Parámetro	*Resultado*	*Clase*
≠ ⊥	≠ ⊥	Total
	⊥	Parcial
⊥	≠ ⊥	No estricta
	⊥	Estricta

Veamos ejemplos de estos casos, empezando con la primera clasificación.

- ```
  succ :: Int -> Int
  succ x = x+1
  ```
- ```
  recip :: Float -> Float
  recip x | x /= 0 = 1/x
  ```

Para toda expresión `n` con valor definido, `succ n` está definida también. La función `succ` es total.

Hay valores definidos para los cuales la función `recip` no está definida. La función `recip` es parcial.

Parece más fácil trabajar con funciones totales, ¿por qué no cambiamos la definición de los tipos de datos para que coincidan con el dominio de nuestras funciones? Esto convertiría todas las funciones en totales. Por ejemplo, podríamos crear un tipo que representara a los `Float` sin el 0 y usarlo como tipo de origen de nuestra función `recip`. Lo malo es que, si extendemos el sistema de tipos para que nos permita definir solamente funciones totales, no se va a poder construir un programa que haga el chequeo de consistencia de tipos en todos los casos y nunca dé ⊥.

Este es un resultado teórico de la llamada teoría de la computación o teoría de la computabilidad. No es tan difícil comprender la demostración, pero escapa a los alcances de nuestra materia. Igual les voy a dar una idea de por qué pasa esto.

Seguramente saben que hay algunos resultados no probados en matemática que se llaman conjeturas. Por mucho tiempo, el llamado “último teorema de Fermat” estuvo en esta categoría. Dice que no hay ningún grupo de cuatro enteros mayores que dos que cumplan la ecuación $x^n+y^n=z^n$.

Hace poco se demostró que es verdad. Pero quedan muchas conjeturas sin demostrar. Una de las más sencillas de expresar es la de Goldbach, que dice que todo entero par mayor que dos es la suma de dos primos. Podríamos definir una función `sirveG` (la dejo como ejercicio) que dijera si su parámetro es o no la suma de dos primos:

```
sirveG :: Int -> Bool
```

Ahora defino otra función de la siguiente manera:

```
sirveGB n = if sirveG n then True else bottom
```

Si yo quisiera que esta función fuera total, tendría que definir un tipo, llamémoslo `Goldbach`, que tuviera todos los pares que fueran suma de dos primos y ninguno que no lo fuera. Para saber si ese tipo contiene o no todos los pares (o los mayores que dos), el intérprete tendría que probar la conjetura de Goldbach, que no está demostrada y tal vez no pueda demostrarse. Lo logre o no lo logre, le va a llevar un buen tiempo (a los matemáticos ya les llevó 260 años, sin éxito), lo cual hace poco útil nuestra máquina de inferencia de tipos de la que esperamos que nos diga rápido si nuestro programa es correcto.

Miren ahora dos versiones de la función que da el inverso de un número:

- ```
  recip :: Float -> Float
  recip x | x /= 0 = 1/x
  ```
- ```
  recipE :: Float -> Float
  recipE x | x /= 0 = 1/x
  recipE x | x == 0 = error "No puedo calcular 1/0"
  ```

En términos denotacionales, las dos son exactamente la misma función parcial: para todo `x`, es `recip x` = `recipE x`. En particular, `recip 0` = `recipE 0` = ⊥. La única diferencia está en el mensaje de error. La primera muestra un mensaje provisto por la implementación del lenguaje, que va a ser más difícil de comprender por parte de alguien no entrenado. Pero ambas son formas perfectamente válidas de implementar una función parcial. El programador que use estas funciones debería conocer las restricciones que tienen y nunca invocarlas pasándoles 0 como parámetro.

Paso a la segunda clasificación, la que separa las funciones en estrictas y no estrictas, de acuerdo a qué resultado dan cuando uno de sus parámetros es indefinido. Recordemos que son estrictas las que al recibir un valor indefinido devuelven también ⊥. Las funciones `succ` y `recip` son estrictas: si reciben un parámetro indefinido no pueden dar un valor como resultado (traten de calcular `succ bottom` o `recip bottom`). Defino otra función:

```
const :: a -> b -> a
const x y = x
```

Esta función recibe dos parámetros y devuelve el primero de ellos. ¿Qué valor tienen las siguientes expresiones?

- `const infinito 2`
- `const 2 infinito`

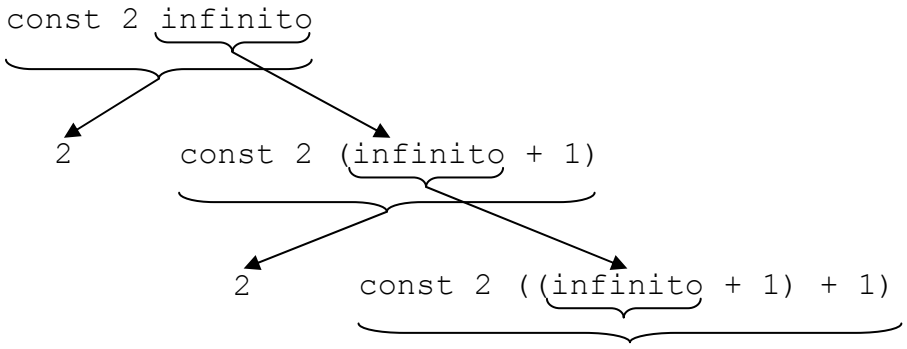
La primera vale ⊥. Pero la segunda puede dejarnos dudas. La función `const` no necesita calcular su segundo argumento para devolver un valor.

Cuando una función recibe ⊥ como uno de los valores necesarios para calcular su resultado, tiene que ser estricta, porque al tratar de calcularlo se va a indefinir. Pero si el parámetro indefinido no se usa en el cálculo, quien diseñe el lenguaje puede decidir qué va a pasar. El secreto está en el orden de evaluación.

Orden de evaluación

Acuérdense de que el orden de evaluación es simplemente una forma de elegir cuál va a ser el próximo redex a reemplazar cuando la expresión tiene más de uno. La confluencia, de la que hablé antes, nos garantiza que si por

dos órdenes de evaluación llegamos a valores definidos, se va a tratar del mismo valor. No obstante, podría ser que por un orden llegáramos a un valor definido y por otro a bottom.



En el gráfico se ve que si en cualquier momento de la reducción se elige como redex toda la expresión se llega a un valor definido; mientras que si se elige como redex la expresión `infinito` se obtiene una expresión más larga.

Los órdenes entre los que optamos en cada paso del ejemplo tienen nombres. El **orden aplicativo** reduce siempre primero los redexes internos (para evaluar una aplicación de función se reemplazan primero los argumentos y después la aplicación). El **orden normal** reduce primero los redexes externos (primero se reemplaza la aplicación y después, si siguen estando, los argumentos). En ambos casos, si hay más de un redex del nivel correspondiente, se elige el de más a la izquierda (primer parámetro no normalizado todavía).

El orden normal lleva ese nombre porque encuentra la forma normal siempre que exista (como en el ejemplo). Reduciendo con orden aplicativo, `const 2 infinito` vale ⊥. Reduciendo con orden normal, vale `2`.

Relacionemos los órdenes con la clasificación de funciones. Usando el orden aplicativo la función `const` es estricta. Con el orden normal es no estricta. De hecho, si el orden de evaluación es aplicativo, todas las funciones son estrictas. Mientras que reduciendo en forma normal hay funciones estrictas y no estrictas (las que no siempre usan todos sus parámetros para calcular su resultado).

En los lenguajes con efectos colaterales (sin transparencia referencial), la evaluación de una expresión puede cambiar el estado de la máquina y por lo tanto el valor de las próximas expresiones a evaluar. En ellos se usa el orden aplicativo para que todas las funciones sean estrictas, lo cual las hace más predecibles. Veamos un ejemplo con efectos colaterales (esto que voy a escribir no es funcional puro, no es Haskell):

```
test :: Int -> Int -> Int
test a b = if (print 2) == 2 then a+b else a+b
```

Se supone que `print x` muestra el valor de `x` y también lo devuelve como resultado. Con el orden aplicativo, la expresión `test 2 (print 1)` imprimiría `1 2 3`. Con el orden normal, imprimiría `2 1 3`.

En un lenguaje puro, el orden de aplicación y, por lo tanto, la posibilidad de definir funciones no estrictas es una decisión de diseño. Los diseñadores de Haskell decidieron que contara con funciones no estrictas, por lo que optaron por el orden normal.

Evaluación lazy

El orden normal a veces ahorra mucho trabajo de computación. Miren estas definiciones:

- ```
  fd :: (Int, Int) -> Int
  fd (a,b) = a+a
  ```
- ```
  test :: (Int, Int)
  test = (3, fib 20)
  ```

Sabiendo que `fib 20` requiere unas 200.000 reducciones, ¿cuántas se requerirán para reducir `fd test` en el orden aplicativo? Muchas. Primero hay que reducir `test` hasta llegar a una expresión normal, lo cual es muy caro. En cambio, reduciendo en orden normal no se calcula `fib 20`, ya que no hace falta para llegar a un resultado. Y se llega en menos de diez reducciones.

Ahora bien, el orden normal no siempre conduce a una evaluación muy eficiente. Fíjense en este otro ejemplo:

```
quin :: Int -> Int
quin x = x+x+x+x+x
```

¿Cuántas reducciones se necesitan para calcular `quin (fib 20)` con el orden aplicativo? Una vez que se hagan las 200.000 reducciones de `fib 20`, se obtiene un número que se suma cinco veces, y listo. ¿Qué pasa con el orden normal? Se reduciría primero a
`(fib 20)+(fib 20)+(fib 20)+(fib 20)+(fib 20)`
¡y faltarían un millón de reducciones!

Por eso, los diseñadores de Haskell optaron por una forma de evaluación que se llama **evaluación lazy** (perezosa, fiaca). Este algoritmo aprovecha la transparencia referencial para “acordarse” del resultado de cada expresión evaluada. Si la expresión vuelve a aparecer, se la reemplaza por el resultado ya calculado. En Haskell, la evaluación de la expresión `quin (fib 20)` va a necesitar más o menos la misma cantidad de reducciones que usando orden aplicativo, porque `fib 20` se va a evaluar una sola vez.

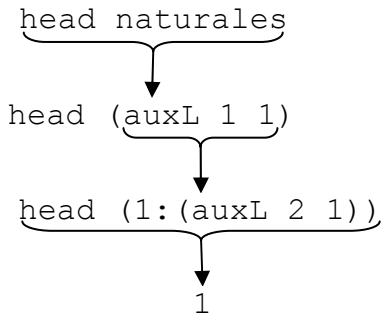
Además de la ventaja de mayor eficiencia en tiempo y espacio, la evaluación lazy agrega a los lenguajes la posibilidad de manipular estructuras de datos y computaciones infinitas.

- ```
  naturales, pares, impares :: [Int]
  naturales = auxL 1 1
  impares = auxL 1 2
  pares = auxL 2 2
  ```
- ```
  auxL :: Int -> Int -> [Int]
  auxL n p = n : auxL (n+p) p
  ```

Si le pidiéramos a un intérprete que evaluara la expresión `naturales`, escribiría algo como lo siguiente:

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,3
0,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,
57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83
,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107
,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127
,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147
,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167
,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187
,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207
,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227
,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247
,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267
,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,286,287
,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,307
,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327
,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347...
```

Y seguiría hasta que nosotros lo paráramos. Pero si en lugar de eso le pidiéramos que evaluara `head naturales`, daría `1` y terminaría. Se debe a que la evaluación lazy no reduce redexes innecesarios. Veamos la reducción:



Y si quisiéramos los diez primeros pares, podríamos usar la función `take` (que toma los primeros *n* elementos de una lista) para pedir `take 10 pares`. Se imprimiría la lista `[2,4,6,8,10,12,14,16,18,20]`.

Las estructuras son infinitas, pero podemos manipularlas usando funciones que no requieren calcularlas completas. Estas listas infinitas de enteros son tan comunes, que Haskell tiene una notación especial para construirlas. La lista que nosotros llamamos `naturales` se puede escribir como `[1..]`; la que llamamos `pares`, como `[2,4..]` y la que llamamos `impares`, como `[1,3..]`.

Acá tienen otros ejemplos. Primero voy a definir una función `sumar2` que suma dos listas de enteros (de igual longitud) elemento a elemento.

- ```
  sumar2::[Int] -> [Int] -> [Int]
  sumar2 [] [] = []
  sumar2 (n:ns) (m:ms) = n+m : sumar2 ns ms
  ```

Y acá vienen las listas infinitas:

- ```
  unos :: [Int]
  unos = auxL 1 0
  ```
- ```
  naturales2 :: [Int]
  naturales2 = 1 : sumar2 unos naturales2
  ```
- ```
  fiboList :: [Int]
  fiboList = 1 : 1 : sumar2 fiboList (tail fiboList)
  ```

Si quieren entretenerse un rato, pueden intentar algunas reducciones. Si los van a probar en Haskell, les recomiendo que usen el tipo `Integer` en lugar de `Int`, porque no tiene limitaciones en cuanto al número de dígitos.

Con esto terminamos la introducción a la programación funcional. Acuérdense de que el porcentaje del paradigma que exploramos es mínimo. Vamos a profundizar en clase y no se olviden de leer el capítulo correspondiente del libro.