ST0244 Lenguajes de Programacion
5. Programación funcional

Andrés Sicard Ramírez

Universidad EAFIT

Semestre 2024-1

Preliminares

Convenciones

- La numeración (capítulos, teoremas, figuras, páginas, etc) en estas diapositivas corresponde a la numeración del texto guía [Lee 2017].
- Los ejemplos que incluyen código fuente están en el repositorio del curso.

Introducción

Característica	Programación imperativa	Programación funcional
Asignación de variables	Si	No
Iteración	Si	No
Recursión	Posible	Necesaria
Funciones de orden superior	Posible	Si
Funciones son ciudadanos de primera clase	No	Si
Efectos colaterales	Si	Evita o aisla
Modelo teórico	Máquina de Turing	Cálculo lambda
Ejecución de programas	Ejecución de instrucciones	Evaluación de expresiones

Introducción

Descripción

*«A **side effect** [efecto colateral] introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.» (O'Sullivan, Goerzen y Stewart 2008, pág. 27)*

Introducción

Descripción

> *«A **side effect** [efecto colateral] introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.» (O'Sullivan, Goerzen y Stewart 2008, pág. 27)*

Descripción

Una **función pura** (*pure function*) es una función libre de efectos colaterales (*p. ej.* no produce cambios en objetos mutables ni salida en dispositivos de I/O). Es decir, una función pura

> *«take all their input as explicit arguments, and produce all their output as explicit results.» (Hutton 2016, § 10.1)*

Introducción

Ejemplo

Mirar los archivos `fp/side-effect-state*`

- Pascal

```
$ cd fp
$ make side-effect-state-pascal
$ ./side-effect-state-pascal
```

- C++

```
$ cd fp
$ make side-effect-state-c-plusplus
$ ./side-effect-state-c-plusplus
```

Introducción

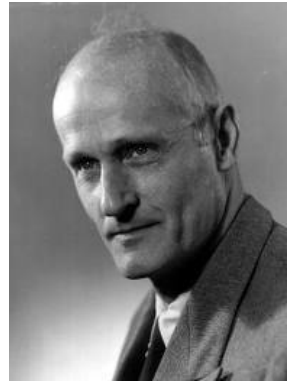
Ejemplo

Mirar el archivo `fp/side-effect-undefined-operation.cc`

```
 $ cd fp
 $ make side-effect-undefined-operation
./side-effect-undefined-operation
```

Cálculo lambda

Alonzo Church (1903–1995)



Stephen Cole Kleene (1909–1994)*

*Fotos tomadas de Wikipedia.

Cálculo lambda

- Un sistema formal creado por Alonzo Church y Stephen Kleene alrededor de 1930.
- El objetivo inicial fue usar el cálculo lambda para las fundaciones de la matemáticas.
- Empleado para estudiar funciones y recursividad.
- Modelo de computabilidad.
- Lenguaje de programación funcional sin tipos.
- Notación (*p. ej.* funciones anónimas y curryficación).

Cálculo lambda

Descripción

En el cálculo lambda hay tres nociones primitivas: **variables**, **abstracción funcional** (*functional abstraction*) y **aplicación funcional** (*functional application*). Además, hay reglas para manipular las expresiones del sistema.

Cálculo lambda

Descripción

En el cálculo lambda hay tres nociones primitivas: **variables**, **abstracción funcional** (*functional abstraction*) y **aplicación funcional** (*functional application*). Además, hay reglas para manipular las expresiones del sistema.

Ejemplo (informal)

En el tablero.

Ejemplo (curryficación)

En el tablero.

Cálculo lambda

Definición

El conjunto de **λ-términos** es definido por la siguiente BNF:

$$
\begin{aligned}
M, N ::= {} & x && \text{(variable)} \\
\mid {} & (\lambda x.M) && \text{(λ-abstracción)} \\
\mid {} & (MN) && \text{(aplicación)}
\end{aligned}
$$

Cálculo lambda

Definición

El conjunto de **λ-términos** es definido por la siguiente BNF:

$$
\begin{aligned}
M, N ::= {} & x && \text{(variable)} \\
\mid {} & (\lambda x.M) && \text{(λ-abstracción)} \\
\mid {} & (MN) && \text{(aplicación)}
\end{aligned}
$$

Convenciones

- Las variables serán denotadas por $x, y, z, \ldots$.
- Los λ-términos serán denotados por $M, N, \ldots$.

Cálculo lambda

Definición

El conjunto de **λ-términos** es definido por la siguiente BNF:

$$
\begin{aligned}
M, N ::= {} & x && \text{(variable)} \\
\mid {} & (\lambda x.M) && \text{(λ-abstracción)} \\
\mid {} & (MN) && \text{(aplicación)}
\end{aligned}
$$

Convenciones

- Las variables serán denotadas por $x, y, z, \ldots$.
- Los λ-términos serán denotados por $M, N, \ldots$.

Ejemplo

En el tablero.

Cálculo lambda

Convenciones y definiciones auxiliares

- Los paréntesis más externos no son escritos.
- La aplicación tiene mayor precedencia, es decir,

$$\lambda x.MN := (\lambda x.(MN)).$$

- La aplicación asocia a la izquierda, es decir,

$$MN_1N_2 \dots N_k := (\dots((MN_1)N_2) \dots N_k).$$

- La λ-abstracción asocia a la derecha, es decir,

$$\begin{aligned} \lambda x_1x_2 \dots x_n.M &:= \lambda x_1.\lambda x_2. \dots . \lambda x_n.M \\ &:= (\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M) \dots))). \end{aligned}$$

Cálculo lambda

Descripción

El comportamiento funcional del λ-cálculo es formalizado vía sus reglas de reducción/conversión.

Definición

La regla de **β-reducción** es definida por

$$(\lambda x.M)N \Rightarrow M[\, x \mapsto N\,],$$

donde $M[\, x \mapsto N\,]$ denota el resultado de **reemplazar cada ocurrencia libre de x en M por N.**[*]

[*] Véase, *p. ej.* [Barendregt 2004; Hindley y Seldin 2008].

Cálculo lambda

Definición

La regla de **β-reducción** es definida por

$$(\lambda x.M)N \Rightarrow M[\, x \mapsto N\,],$$

donde $M[\, x \mapsto N\,]$ denota el resultado de **reemplazar cada ocurrencia libre de x en M por N.**[*]

Ejemplo

En el tablero.

[*] Véase, *p. ej.* [Barendregt 2004; Hindley y Seldin 2008].

Cálculo lambda

Definición

Un **redex** es un λ-término de la forma $(\lambda x.M)N$.

Definición

Un λ-término el cual no contiene ningún redex está en **forma normal**.

Cálculo lambda

Definición

Un **redex** es un λ-término de la forma $(\lambda x.M)N$.

Definición

Un λ-término el cual no contiene ningún redex está en **forma normal**.

Ejemplo

En el tablero.

Cálculo lambda

Definición

Un redex es un ***outermost* redex** sii éste no está contenido en en otro redex.

Un redex es un ***innermost* redex** sii no contiene otro redex.

Ejemplo

Sea $M := (\lambda y.z)((\lambda x.xx)(\lambda x.xx))$. Entonces

- M es un *outermost* redex.
- M no es un *innermost* redex porque contiene el redex $(\lambda x.xx)(\lambda x.xx)$.
- $(\lambda x.xx)(\lambda x.xx)$ es un *innermost* redex $(\lambda x.xx)(\lambda x.xx)$.
- $(\lambda x.xx)(\lambda x.xx)$ no es un *outermost* porque está contenido en el redex M.

Cálculo lambda

Definición

La ***normal order reduction*** es una estrategia de evaluación en la cual el *left-most outermost* redex es reducido primero.

Cálculo lambda

Definición

La ***normal order reduction*** es una estrategia de evaluación en la cual el *left-most outermost* redex es reducido primero.

Definición

La ***applicative order reduction*** es una estrategia de evaluación en la cual el *left-most innermost* redex es reducido primero.

Cálculo lambda

Ejemplo

Reducir $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)$ usando ambas estrategias de evaluación.

Normal order reduction

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)$$
$$\Rightarrow \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)}$$
$$\Rightarrow \lambda z.\underline{(\lambda x.x)z}((\lambda xy.x)z)$$
$$\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})$$
$$\Rightarrow \lambda z.z(\lambda y.z)$$

Applicative order reduction

$$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)$$
$$\Rightarrow (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda xy.x)$$
$$\Rightarrow \underline{(\lambda yz.z(yz))(\lambda xy.x)}$$
$$\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})$$
$$\Rightarrow \lambda z.z(\lambda y.z)$$

Cálculo lambda

Ejemplo

Sea $\Omega := (\lambda x.xx)(\lambda x.xx)$. Reducir $(\lambda y.z)\Omega$ usando ambas estrategias de evaluación.

Normal order reduction

$$\underline{(\lambda y.z)\Omega}$$
$$\Rightarrow z$$

Applicative order reduction

$$(\lambda y.z)\Omega$$
$$= (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})$$
$$\Rightarrow (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})$$
$$\Rightarrow (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})$$
$$\Rightarrow \ldots$$

Observación

Church [1935, 1936] demostró que el conjunto

$$\{\, M \in \text{λ-términos} \mid M \text{ tiene forma normal} \,\}$$

es indecidible. Éste fue el primer conjunto indecible.

Introducción a Haskell

Características importantes*

- Haskell es un lenguaje de programación funcional pura con evaluación perezosa (*lazy*).
- Haskell suporta funciones de orden superior y las funciones son ciudadanos de primera clase.
- Haskell es fuertemente tipeado tal como lo es Pascal, pero su sistema de tipos es más potente ya que éste soporta *polymorphic type checking*.

  Observación a la afirmación:

  «With this strong type checking it is pretty infrequent that you need to debug your code!! What a great thing!!!» (pág. 184)

(continua en la próxima diapositiva)

*Adaptación de la sección "5.3 Getting Started with Standard ML" del texto guía.

Características importantes (continuación)

- Haskell proporciona un entorno seguro para el desarrollo y ejecución de programas. No hay apuntadores en Haskell.
- Desde que no hay apuntadores, la recolección de basura (*garbage collection*) es implementada en el sistema de tiempo de ejecución del lenguaje.
- El lenguaje ofrece ajuste de patrones (*pattern-matching*) para la escritura de funciones recursivas.
- Listas son un tipo de dato *built-in*.
- Una biblioteca de estructuras de datos y funciones frecuentemente usadas está disponible. Esta biblioteca es llamada la *base library*.

Lectura sugerida

Hughes [1989, pág. 107] escribió:

> *«In this paper, we have argued that modularity is the key to successful programming [...] Functional programming languages provide two new kinds of glue—higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways [...] Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones.»*

Lectura sugerida

Hughes [1989, pág. 107] escribió:

> *«In this paper, we have argued that modularity is the key to successful programming [...] Functional programming languages provide two new kinds of glue—higher-order functions and lazy evaluation. Using these glues one can modularise programs in new and exciting ways [...] Smaller and more general modules can be re-used more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones.»*

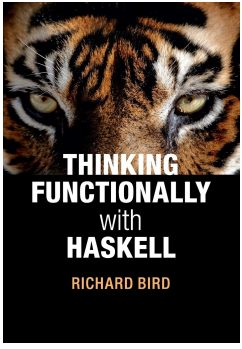
Observación

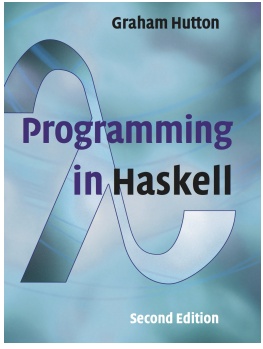
El artículo fue escrito en 1984 y circuló informalmente. El artículo no usó Haskell sino Miranda, un lenguaje predecesor de Haskell.

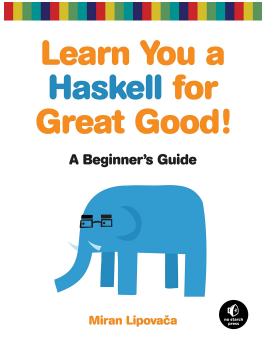
Historia

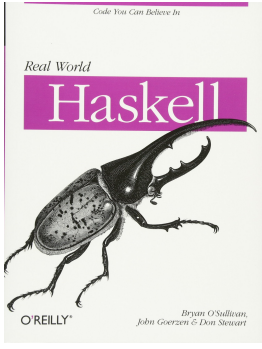
Haskell fue definido por un comité. El «Haskell version 1.0 Report» fue publicado el primero de abril de 1990. Una historia muy completa es presentada en [Hudak, Hughes, Peyton Jones y Wadler 2007].

Algunos libros









Introducción a Haskell

Usando Haskell

- Compilador e interpretador: The Glorious Glasgow Haskell Compilation System (GHC)
  - Compilador: `ghc`
  - Interpretador: `ghci`
- Para instalar GHC sugerimos usar `ghcup`.
- Para instalar bibliotecas o programas, y para compilar programas usted puede usar `Cabal` o `stack`.
- Hackage: El repositorio de paquetes (programas y bibliotecas) para Haskell.

Ejemplo

La función factorial.

```
1 fac1 :: Int -> Int
2 fac1 n = product [1..n]
```

Expresiones, tipos y funciones

Ejemplo

La función factorial.

```
1 fac1 :: Int -> Int
2 fac1 n = product [1..n]
```

Pregunta

¿Está la función `fac1` bien definida? ¿Qué acerca de `fac1 (-2)`?

Expresiones, tipos y funciones

Ejemplo

La función factorial.

```
1 fac1 :: Int -> Int
2 fac1 n = product [1..n]
```

Pregunta

¿Está la función `fac1` bien definida? ¿Qué acerca de `fac1 (-2)`?

```
1 import Numeric.Natural (Natural)
2
3 fac2 :: Natural -> Natural
4 fac2 n = product [1..n]
```

Expresiones, tipos y funciones

Otras implementaciones de la función factorial (humor)

Buscar «The evolution of a Haskell programmer» en Internet.

Curryficación

Funciones para curryficar y descurryficar

(i) Convierte un función no curryficada en una función curryficada.

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

(ii) Convierte una función curryficada en un función no curryficada.

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

Listas

Definición inductiva

Haskell tiene sintaxis built-in para listas, donde una lista es:

- la lista vacía, escrita `[]`, o
- un primer elemento `x` y una lista `xs`, escrita `(x : xs)`.

El operador «`:`» es llamado *cons* usualmente.

Listas

Ejemplo (función recursiva usando ajuste de patrones en listas)

Retorna la longitud de una lista de `Int`s.

```
1 lengthI :: [Int] -> Int
2 lengthI []       = 0
3 lengthI (x : xs) = 1 + lengthI xs
```

Listas

Ejemplo (función recursiva usando ajuste de patrones en listas)

Retorna la longitud de una lista de `Int`s.

```
1 lengthI :: [Int] -> Int
2 lengthI []       = 0
3 lengthI (x : xs) = 1 + lengthI xs
```

Pregunta

¿Qué acerca de una función similar sobre una lista de booleanos?

Listas

Ejemplo (función recursiva usando ajuste de patrones en listas)

Retorna la longitud de una lista de `Int`s.

```
1 lengthI :: [Int] -> Int
2 lengthI []       = 0
3 lengthI (x : xs) = 1 + lengthI xs
```

Pregunta

¿Qué acerca de una función similar sobre una lista de booleanos?

```
1 lengthB :: [Bool] -> Int
2 lengthB []       = 0
3 lengthB (x : xs) = 1 + lengthB xs
```

Listas

Pregunta

¿Podemos evitar la duplicación del código (*boilerplate code*)? ¡Si!

Polimorfismo paramétrico

Listas

Las listas *built-in* están definidas usando polimorfismo paramétrico.

```
GHCi> :t []
[] :: [a]

GHCi> :t (:)
(:) :: a -> [a] -> [a]
```

Polimorfismo paramétrico

Ejemplo

Retorna la longitud de una lista finita de cualquier tipo.

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

Polimorfismo paramétrico

Ejemplo

Concatenación de dos listas.

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

Polimorfismo paramétrico

Ejemplo (algunas funciones en la *base library*)

- Retorna el primer elemento de una lista no vacía.

```
head :: [a] -> a
```

- Retorna el último elemento de una lista finita y no vacía.

```
last :: [a] -> a
```

- Retorna la cola de una lista no vacía.

```
tail :: [a] -> [a]
```

Polimorfismo paramétrico

Ejemplo (algunas funciones en la *base library*)

- Retorna todos los elementos excepto el último de una lista no vacía.

```
init :: [a] -> [a]
```

- Determina si una lista está o no vacía.

```
null :: [a] -> Bool
```

Recursión

Definición

Una función es **recursiva** sii ésta se llama a si misma.

Recursión

Escribiendo funciones recursivas (pág. 188)

1. «*Decide what the function is named, what arguments are passed to it, and what the function should return.*»
2. «*At least one of the arguments must get smaller each time. Most of the time it is only one argument getting smaller. Decide which one that will be.*»
3. «*Write the function declaration, declaring the name, arguments types, and return type if necessary.*»
4. «*Write a base case for the argument that you decided will get smaller. Pick the smallest, simplest value that could be passed to the function and just return the result for that base case.*»
5. «*The next step is the crucial step. You don't write the next statement from left to right. You write from the inside out at this point.*»

(continua en la próxima diapositiva)

Recursión

Escribiendo funciones recursivas (pág. 188) (continuación)

6. «*Make a recursive call to the function with a smaller value. For instance, if it is a list you decided will get smaller, call the function with the tail of the list. If an integer is the argument getting smaller, call the function with the integer argument minus 1. Call the function with the required arguments and in particular with a smaller value for the argument you decided would get smaller at each step.*»
7. «*Now, here's a leap of faith. That call you made in the last step worked! It returned the result that you expected for the arguments it was given. Use that result in building the result for the original arguments passed to the function. At this step it may be helpful to try a concrete example. Assume the recursive call worked on the concrete example. What do you have to do with that result to get the result you wanted for the initial call? Write code that uses the result in building the final result for your concrete example. By considering a concrete example it will help you see what computation is required to get your final result.*»
8. «*That's it! Your function is complete and it will work if you stuck to these guidelines.*»

Recursión

Ejemplo

Las funciones anteriores son recursivas:

```
factorial :: Int -> Int
length    :: [a] -> Int
(++)      :: [a] -> [a] -> [a]
last      :: [a] -> a
init      :: [a] -> [a]
null      :: [a] -> Bool
```

Caracteres y cadenas

En Haskell el tipo de los caracteres es `Char` y el tipo `String` es un *type synonymous* de `[Char]`. Es decir, es una cadena es una lista de caracteres.

Caracteres y cadenas

En Haskell el tipo de los caracteres es `Char` y el tipo `String` es un *type synonymous* de `[Char]`. Es decir, es una cadena es una lista de caracteres.

Ejemplo

```
'a'                   -- Character.
'a' : 'b' : 'c' : []  -- List of characters.
['a','b','c']         -- List of characters.
"abc"                 -- String.

-- List of strings.
["hello","how"] ++ ["are","you?"]
```

Evaluación perezosa

Descripción

En la **evaluación perezosa** (*lazy evaluation*) los parámetros de las funciones no son evaluados hasta que sea necesario.

Evaluación perezosa

Ejemplo

```
1 -- Disjunction
2 (||) :: Bool -> Bool -> Bool
3
4 foo :: Int -> Bool  -- Non-terminating funcion
5 foo n = foo (n + 1)
6
7 bar :: Int -> Bool
8 bar n = True || foo n
```

Evaluación perezosa

Ejemplo

```
1 -- Disjunction
2 (||) :: Bool -> Bool -> Bool
3
4 foo :: Int -> Bool  -- Non-terminating funcion
5 foo n = foo (n + 1)
6
7 bar :: Int -> Bool
8 bar n = True || foo n
```

Pregunta

¿Cuál es el valor de `bar 10`?

Evaluación perezosa

Ejemplo

```
1 -- Disjunction
2 (||) :: Bool -> Bool -> Bool
3
4 foo :: Int -> Bool  -- Non-terminating funcion
5 foo n = foo (n + 1)
6
7 bar :: Int -> Bool
8 bar n = True || foo n
```

Pregunta

¿Cuál es el valor de `bar 10`? `True`.

Evaluación perezosa

Ejemplo (from http://stackoverflow.com/questions/30688558/)

```
1 dh :: Int -> Int -> (Int, Int)
2 dh d q = (2^d, q^d)
3
4 a :: (Int, Int)
5 a = dh 2 (fst b)
6
7 b :: (Int, Int)
8 b = dh 3 (fst a)
```

Evaluación perezosa

Ejemplo (from http://stackoverflow.com/questions/30688558/)

```
1 dh :: Int -> Int -> (Int, Int)
2 dh d q = (2^d, q^d)
3
4 a :: (Int, Int)
5 a = dh 2 (fst b)
6
7 b :: (Int, Int)
8 b = dh 3 (fst a)
```

Pregunta

¿Cuál es el valor de a?

Evaluación perezosa

Ejemplo (from http://stackoverflow.com/questions/30688558/)

```
1 dh :: Int -> Int -> (Int, Int)
2 dh d q = (2^d, q^d)
3
4 a :: (Int, Int)
5 a = dh 2 (fst b)
6
7 b :: (Int, Int)
8 b = dh 3 (fst a)
```

Pregunta

¿Cuál es el valor de a? `(4,64)`.

Evaluación perezosa

Ejemplo

La expresión `take n` aplicada a una lista `xs` retorna el prefijo `xs` de longitud `n`, `xs` is `n > length xs`.

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

Lista no acotada.

```
1 ones :: [Int]
2 ones = 1 : ones
```

Evaluación perezosa

Ejemplo

La expresión `take n` aplicada a una lista `xs` retorna el prefijo `xs` de longitud `n`, `xs is n > length xs`.

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

Lista no acotada.

```
1 ones :: [Int]
2 ones = 1 : ones
```

Pregunta

¿Cuál es el valor de `take 5 ones`?

Evaluación perezosa

Ejemplo

La expresión `take n` aplicada a una lista `xs` retorna el prefijo `xs` de longitud `n`, `xs is n > length xs`.

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

Lista no acotada.

```
1 ones :: [Int]
2 ones = 1 : ones
```

Pregunta

¿Cuál es el valor de `take 5 ones`? `[1,1,1,1,1]`.

Tipos de datos algebraicos

Ejemplo

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

`True` y `False` son los (data) constructores del tipo de datos `Bool`.

Tipos de datos algebraicos

Ejemplo

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

`True` y `False` son los (data) constructores del tipo de datos `Bool`.

Ejemplo (función por ajuste de patrones)

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

Tipos de datos algebraicos

Ejemplo

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Función por ajuste de patrones.

```
1 nextDay :: Day -> Day
2 nextDay Mon = Tue
3 nextDay Tue = Wed
4 nextDay Wed = Thu
5 nextDay Thu = Fri
6 nextDay Fri = Sat
7 nextDay Sat = Sun
8 nextDay Sun = Mon
```

Tipos de datos algebraicos

Ejemplo (tipo de dato recursivo)

```
data Nat = Zero | Succ Nat
```

Tipos de datos algebraicos

Ejemplo (tipo de dato recursivo)

```
data Nat = Zero | Succ Nat
```

Ejemplo (función (estructuralmente) recursiva)

```
1 (+) :: Nat -> Nat -> Nat
2 Zero     + n = n
3 (Succ m) + n = Succ (m + n)
```

Tipos de datos algebraicos

Ejemplo (tipo de dato polimórfico)

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

Tipos de datos algebraicos

Ejemplo (tipo de dato polimórfico)

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

```
-- Las listas built-in.
data [] a = [] | a : [a]
```

Tuples

Ejemplo

Mirar el archivo `fp/Tuples.hs`.

Expresiones **let** y cláusulas **where**

Ejemplo

Mirar el archivo `fp/LetWhere.hs`.

Expresiones **let** y cláusulas **where**

Ejemplo

Mirar el archivo `fp/LetWhere.hs`.

Ejemplo

Tomado de [Hudak, Peterson y Fasel 1999].

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

- Las ligaduras creadas por las expresiones **let** son mutuamente recursivas.
- Las declaraciones permitidas en las expresiones **let** incluyen tipos y funciones.

Expresiones **let** y cláusulas **where**

Ejemplo

Tomado de [Hudak, Peterson y Fasel 1999].

```
1  f x y  |  y > z   =  ...
2         |  y == z  =  ...
3         |  y < z   =  ...
4    where z = x * x
```

- Una cláusula **where** es parte de la sintaxis de la definición de la función.
- En este caso, no podemos reemplazar la cláusula **where** por una expresión **let**.

Eficiencia de la recursión

Ejemplo (función de Fibonnaci)

Versión muy ineficiente.

```
1    fib :: Natural -> Natural
2    fib 0 = 0
3    fib 1 = 1
4    fib n = fib (n - 1) +
5            fib (n - 2)
```

El número de llamadas a `fib` crece exponencialmente con el tamaño de `n`.

`fib 4`: 9 llamadas
`fib 5`: 15 llamadas
`fib 6`: 15 + 9 llamadas

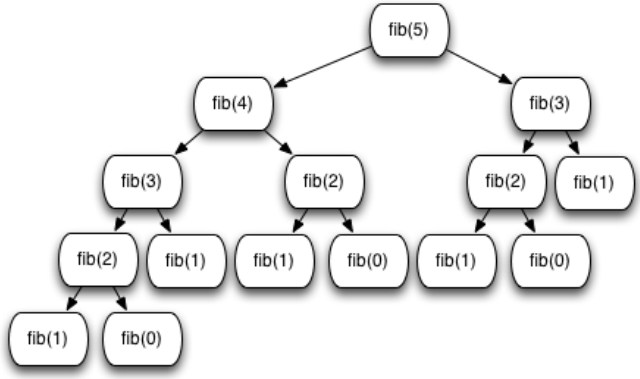


Fig. 5.16.

(continua en la próxima diapositiva)

Eficiencia de la recursión

Ejemplo (continuación)

Versión usando un patrón de acumulación (*accumulator pattern*).

```
1 fibAP :: Natural -> Natural
2 fibAP n =
3   let fibH :: Natural -> Natural -> Natural -> Natural
4       fibH count current previous =
5         if count == n then previous
6         else fibH (count + 1) (current + previous) current
7   in  fibH 0 1 0
```

(continua en la próxima diapositiva)

Eficiencia de la recursión

Ejemplo (continuación)

```
fibAP 5 = fibH 0 1 0
        = fibH 1 1 1
        = fibH 2 2 1
        = fibH 3 3 2
        = fibH 4 5 3
        = fibH 5 8 5
        = 5
```

$$\begin{aligned}
&\text{fib} \quad : \mathbb{N} \to \mathbb{N} \\
&\text{fib}(0) = 0, \\
&\text{fib}(1) = 1, \\
&\text{fib}(2) = 1, \\
&\text{fib}(3) = 2, \\
&\text{fib}(4) = 3, \\
&\text{fib}(5) = 5.
\end{aligned}$$

(continua en la próxima diapositiva)

Eficiencia de la recursión

Ejemplo (continuación)

Mirar los archivos `Fib1.hs` y `Fib2.hs` en el directorio `fp/acummulator-pattern/fibonacci`.

(i) Tiempo de ejecución para la versión ineficiente

```
$ make fib1
$ time ./fib1
real      1m4.353s
user      1m4.160s
sys       0m0.192s
```

(ii) Tiempo de ejecución para la versión empleando el patrón de acumulación.

```
$ make fib2
$ time ./fib2
real      0m0.006s
user      0m0.001s
sys       0m0.005s
```

Eficiencia de la recursión

Ejemplo

(i) Reverso de una lista usando concatenación

```
1 reverse :: [a] -> [a]
2 reverse []       = []
3 reverse (x : xs) = reverse xs ++ [x]
```

Eficiencia de la recursión

Ejemplo

(i) Reverso de una lista usando concatenación

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

(ii) Reverso de una lista usando el patrón de acumulación (*accumulator pattern*).

```haskell
reverse :: [a] -> [a]
reverse xs = rev xs []
  where
   rev []       zs = zs
   rev (y : ys) zs = rev ys (y : zs)
```

Mirar el archivo `fp/Reverse.hs`.

Recursión de cola

Problema

La llamadas recursivas a una función requieren más tiempo de ejecución que un simple ciclo.

Recursión de cola

Problema

La llamadas recursivas a una función requieren más tiempo de ejecución que un simple ciclo.

Solución

Optimización de recursión de cola (*tail recursion*): La implementación de funciones con recursión de cola se realiza eficientemente por el compilador.

Recursión de cola

Problema

La llamadas recursivas a una función requieren más tiempo de ejecución que un simple ciclo.

Solución

Optimización de recursión de cola (*tail recursion*): La implementación de funciones con recursión de cola se realiza eficientemente por el compilador.

Definición

Una función con **recursión de cola (*tail recursion*)** es una función recursiva en donde la última operación es la llamada recursiva a la función.

Recursión de cola

Problema

La llamadas recursivas a una función requieren más tiempo de ejecución que un simple ciclo.

Solución

Optimización de recursión de cola (*tail recursion*): La implementación de funciones con recursión de cola se realiza eficientemente por el compilador.

Definición

Una función con **recursión de cola (*tail recursion*)** es una función recursiva en donde la última operación es la llamada recursiva a la función.

Ejemplo (función factorial)

Mirar el directorio `fp/tail-recursion/factorial`.

Funciones anónimas

Ejemplo

La función anónima $\lambda xy.y^2 + x$ puede ser implementada en Hakell por

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

Las funciones anónimas se aplican de la manera usual.

```
(\ x y -> y * y + x) 3 4
```

También podemos ligar un identificar a una función anónima.

```
foo :: Int -> Int -> Int
foo = (\ x y -> y * y + x)
foo 3 4
```

Funciones de orden superior

Definición

Un función es de **orden superior** sii

i) ésta recibe (al menos) una función como un argumento o

ii) ésta retorna (al menos) una función como resultado.

Funciones de orden superior

Ejemplo

El operador de composición `(.)` compone dos funciones. Éste es definido en la biblioteca base.

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

Mirar el archivo `fp/HigherOrder.hs`.

Funciones de orden superior

Ejemplo

La función `map` aplica una función a cada elemento de una lista.

La expresión `map f xs` es la lista obtenida al aplicar la función `f` a cada elemento de la lista `xs`.

La función `map` esta definida en la biblioteca base.

```
1 map :: (a -> b) -> [a] -> [b]
2 map _ []       = []
3 map f (x : xs) = f x : map f xs
```

Mirar el archivo `fp/HigherOrder.hs`.

Funciones de orden superior

Ejemplo

La función `foldr` sobre listas reduce una lista de derecha a izquierda usando una función binaria, es decir,

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

La función `foldr` sobre listas puede ser definida por

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f z []       = z
3 foldr f z (x : xs) = f x (foldr f z xs)
```

Mirar el archivo `fp/HigherOrder.hs`.

Funciones de orden superior

Ejemplo

La función `foldl` sobre listas reduce una lista de izquierda a derecha usando una función binaria, es decir,

```
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

La función `foldl` sobre listas puede ser definida por

```
1 foldl :: (a -> b -> b) -> b -> [a] -> b
2 foldl f z []       = z
3 foldl f z (x : xs) = foldl f (f z x) xs
```

Funciones de orden superior

Ejemplo

La función `filter` retorna una lista de aquellos elementos de una lista que satisfacen un predicado (es decir, a function `a -> Bool`).

La función `filter` es definida en la biblioteca base.

```
1  filter :: (a -> Bool) -> [a] -> [a]
2  filter _ []       = []
3  filter p (x : xs)
4    | p x           = x : filter p xs
5    | otherwise     = filter p xs
```

Type Classes

Ejemplo

¿Está un elemento en una lista?

```
1 elem :: a -> [a] -> Bool
2 elem x []       = False
3 elem x (y : ys) = x == y || elem x ys
```

Type Classes

Ejemplo

¿Está un elemento en una lista?

```
1 elem :: a -> [a] -> Bool
2 elem x []       = False
3 elem x (y : ys) = x == y || elem x ys
```

El código arriba genera el siguiente error:

```
No instance for (Eq a) arising from a use of ‘==’
```

(continua en la próxima diapositiva)

Type Classes

Ejemplo (continuación)

Podemos arreglar el problema adicionando la **type constraint** `Eq a`, la cual restringe la variable de tipos a únicamente a instancias de la type class `Eq`.

```
1 elem :: Eq a => a -> [a] -> Bool
2 elem x []       = False
3 elem x (y : ys) = x == y || elem x ys
```

Type Classes

Descripción

Las type classes en Haskell proporcionan una forma estructurada para tener polimorfismo ad hoc o sobrecarga de operadores.

Type Classes

Ejemplo

La type class `Eq` está definida por

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

Type Classes

Ejemplo

La type class `Eq` está definida por

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

El tipo de dato `Bool` es una instancia de la *type class* `Eq`.

```
1 instance Eq Bool where
2   True  == True  = True
3   False == False = True
4   _     == _     = False
```

Type Classes

Ejemplo

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
GHCi> elem Mon [Tue, Sat, Sun]
error: No instance for (Eq Day) arising from a use of '=='
```

(continua en la próxima diapositiva)

Type Classes

Ejemplo (continuación)

Una solución: Adicionar la instancia requerida.

```
1  instance Eq Day where
2    Mon == Mon = True
3    Tue == Tue = True
4    Wed == Wed = True
5    Thu == Thu = True
6    Fri == Fri = True
7    Sat == Sat = True
8    Sun == Sun = True
9    _   == _   = False
```

(continua en la próxima diapositiva)

Type Classes

Ejemplo (continuación)

Otra solución: Usar *deriving*

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
         deriving Eq
```

Input and Output

The problem

How can programs with input and output be modelled as pure functions?

Input and Output

The problem

How can programs with input and output be modelled as pure functions?

The solution

There are various approaches for using pure functions and side-effects (see, *p. ej.* [Peyton Jones y Wadler 1993]). Haskell's solution is via *monads*.

Input and Output

The unit type

The unit type is a type with only one element. Haskell unity type and its element are

```
() :: ()
```

The unit type is useful when performing input-output.

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

A program with input-output can be represented by a function

```
World -> World
```

```
type IO = World -> World
```

Input and Output

The `IO` type

The following description of the type `IO` is from [Hutton 2016, § 10.2].

A program with input-output can be represented by a function

```
World -> World
```

```
type IO = World -> World
```

What about if the program returns a value?

```
type IO a = World -> (a, World)
```

(continua en la próxima diapositiva)

Input and Output

The IO type (continuación)

What about if the program requires an argument?

For example, a program requiring a character and returning an integer has the type

```
Char -> IO Int

Char -> World -> (Int, World)
```

Input and Output

The `IO` type (continuación)

What about if the program requires an argument?

For example, a program requiring a character and returning an integer has the type

```
Char -> IO Int

Char -> World -> (Int, World)
```

The compiler has the responsibility of handling the state of world. The type `IO a` is primitive in Haskell.

Input and Output

Definición

An **action** is an expression of type `IO a`. When the expression is evaluated the action is performed.

Input and Output

Definición

An **action** is an expression of type `IO a`. When the expression is evaluated the action is performed.

Ejemplo

- `t : IO Char` is the action that returns a character.
- `t : IO ()` is the action that no returns a value, where `()` is a dummy result value.

Input and Output

The abstract datatype `IO a`

The abstract datatype `IO a` has (at least) the following operations [Bird 1998, § 10.1]:

```
return  :: a -> IO a
(>>=)   :: IO a -> (a -> IO b) -> IO b
putChar :: Char -> IO ()
getChar :: IO Char
```

Input and Output

Ejemplo

Mirar el archivo `fp/IO.hs`.

Testing usando QuickCheck

Un artículo

Claessen, Koen y Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: https://doi.org/10.1145/357766.351266.

*See www.sigplan.org/Awards/ICFP/.

Testing usando QuickCheck

Un artículo

Claessen, Koen y Hughes, John [2000]. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP'00. DOI: https://doi.org/10.1145/357766.351266.

Most Influential ICFP Paper Award 2010*

> *«The techniques described in the paper have spawned a significant body of follow-on work in test case generation. They have also been adapted to other languages . . . »*

*See www.sigplan.org/Awards/ICFP/.

Testing usando QuickCheck

Biblioteca de código abierto

QuickCheck en Hackage.*

*http://hackage.haskell.org/package/QuickCheck.

Testing usando QuickCheck

Biblioteca de código abierto

QuickCheck en Hackage.*

Comercialización

QuviQ (www.quviq.com/).

*http://hackage.haskell.org/package/QuickCheck.

Testing usando QuickCheck

Adaptaciones

La biblioteca QuickCheck ha sido portada a varios lenguajes de programación (Wikipedia 2024-02-02).

C	C#	C++	Chicken	Clojure
Common Lisp	Coq	D	Elm	Elixir
Erlang	F#	Factor	Go	Io
Java	JavaScript	Julia	Logtalk	Lua
Mathematica	Objective-C	OCaml	Perl	Prolog
PHP	Pony	Python	R	Racket
Ruby	Rust	Scala	Scheme	Smalltalk
Standard ML	Swift	TypeScript	Visual Basic .NET	Vhiley

Testing usando QuickCheck

Falsos positivos

El programa está bien implementado pero el test falló.

- Hay un error en alguna parte.
- Hay un error en la especificación.

Testing usando QuickCheck

Falsos positivos

El programa está bien implementado pero el test falló.

- Hay un error en alguna parte.
- Hay un error en la especificación.

Falsos negativos

El programa tiene un error pero pasa el test.

Recordemos la famosa frase de Dijkstra en 1969:

> *«Program testing can be used to show the presence of bugs, but never to show their absence!» (Dijkstra 1970)*

Testing usando QuickCheck

Algunos ejemplos

Referencias


Barendregt, H. P. [1981] (2004). The Lambda Calculus. Its Syntax and Semantics. Revised edition, 6th impression. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier (vid. págs. 18, 19).


Bird, Richard [1988] (1998). Introduction to Functional Programming. 2.ª ed. Prentice Hall Press (vid. pág. 113).


Church, Alonzo (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, págs. 332-333. DOI: 10.1090/S0002-9904-1935-06102-6 (vid. pág. 27).


— (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, págs. 345-363. DOI: 10.2307/2371045 (vid. pág. 27).


Dijkstra, E. W. (1970). Structured Programming. En: Software Engineering Techniques (NATO Software Engineering Conference 1969). Ed. por Buxton, J. N. y Randell, B., págs. 84-88 (vid. págs. 120, 121).


Hindley, J. Roger y Seldin, Jonathan P. (2008). Lambda-Calculus and Combinators. An Introduction. Cambridge University Press (vid. págs. 18, 19).

Referencias

Hudak, Paul, Hughes, John, Peyton Jones, Simon y Wadler, Philip (2007). A History of Haskell: Being Lazy with Class. En: Proceedings of the third ACM SIGPLAN conference on History of programming languages. HOPL III, 12:1-12:55. DOI: 10.1145/1238844.1238856 (vid. pág. 32).

Hudak, Paul, Peterson, John y Fasel, Joseph H. (1999). A Gentle Introduction to Haskell 98. URL: https://www.haskell.org/tutorial/ (vid. págs. 74-76).

Hughes, J. (1989). Why Functional Programming Matters. The Computer Journal 32.2, págs. 98-107 DOI: 10.1093/comjnl/32.2.98 (vid. págs. 30, 31).

Hutton, Graham [2007] (2016). Programming in Haskell. 2.ª ed. Cambridge University Press (vid. págs. 4, 5, 106-108).

Lee, Kent D. [2014] (2017). Foundations of Programming Languages. 2.ª ed. Undergraduate Topics in Computer Science. Springer (vid. pág. 2).

O'Sullivan, Bryan, Goerzen, John y Stewart, Don (2008). Real World Haskell. O'Really Media, Inc. (vid. págs. 4, 5).

Peyton Jones, Simon L. y Wadler, Philip (1993). Imperative Functional Programming. En: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1993), págs. 71-84. DOI: 10.1145/158511.158524 (vid. págs. 103, 104).