ST0244 Lenguajes de Programacion
Programación funcional

Andrés Sicard Ramírez

Universidad EAFIT

Semestre 2024-2

Preliminares

Convenciones

- La numeración (capítulos, teoremas, figuras, páginas, etc) en estas diapositivas corresponde a la numeración del texto guía [Louden y Lambert 2011].
- Los ejemplos que incluyen código fuente están en el repositorio del curso.

Esquema de la presentación

Tema

Introducción a la programación funcional

Scheme

Haskell

Cálculo lambda

Referencias

Introducción a la programación funcional

Característica	**Programación imperativa**	**Programación funcional**
Asignación de variables	Si	No (pura)
Iteración	Si	No (pura)
Recursión	Posible	Necesaria
Modelo	Arquitectura de von Neumann	Cálculo lambda
Ejecución de programas	Ejecución de instrucciones	Evaluación de expresiones
Efectos colaterales	Si	No (pura)
Funciones de orden superior	Posible	Si
Funciones son ciudadanos de primera clase	No	Si

Introducción a la programación funcional

Descripción

Las **variables** en la programación funcional (pura) son tratadas como las variables en matemáticas.

	variable	asignación
programación imperativa	se refiere a una posición en memoria que almacena un valor	cambiar el valor almacenado en un posición en memoria
matemáticas	se refiere a un valor	

Introducción a la programación funcional

Descripción

El **cálculo lambda** es un modelo de computación en donde las funciones son un objeto primitivo:

- definición de funciones
- aplicación de funciones

Introducción a la programación funcional

Descripción

En la programación imperativa hay **instrucciones** y **expresiones**. El propósito de ejecutar una instrucción es modificar el estado global del programa que está siendo ejecutado. En la programación funcional pura solo hay expresiones. El propósito de evaluar una expresión es obtener un valor.

Introducción a la programación funcional

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 a la programación funcional

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)*

Ejemplo (efectos colaterales)

Mirar los archivos en el directorio `fp/side-effects`.

- Pascal

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

- C++

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

Ejemplo (efectos colaterales)

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

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

Recolección de basura

- El recolector de basura (*garbage collector*) remueve automáticamente memoria que fue localizada dinámicamente por el programa pero que ya no es necesitada.
- Conflicto entre el control del programador y la memoria administrada automáticamente.
- El recolector de basura evita pérdidas de memoria.
- El recolector de basura impacta el rendimiento en tiempo de ejecución del programa.
- Lenguajes con recolector de basura requieren un sistema de tiempo de ejecución para ejecutar los programas.

Tema

Introducción a Scheme

Un lenguaje simple pero de propósito general

El primer párrafo de *The R7RS small language specification* menciona que:*

> *«Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.»*

*Véase `https://standards.scheme.org/`.

Introducción a Scheme

Características importantes

- Scheme es un lenguaje de programación funcional (pero soporta la programación imperativa, es decir, tiene efectos colaterales).
- Scheme, al igual que todos los lenguajes funcionales, soporta las funciones de orden superior y las funciones son ciudadanos de primera clase.
- En Scheme los parámetros a las funciones son evaluados antes de ser pasados.
- El chequeo de tipos es Scheme es dinámico.
- Scheme tiene recolector de basura (*garbage collector*).
- TODO Static binding.

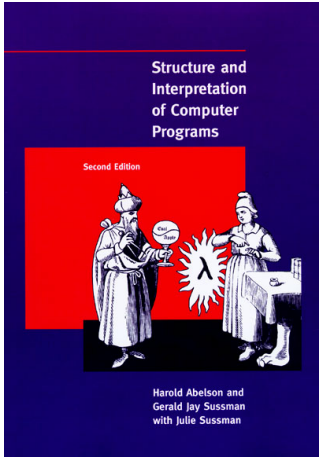
Introducción a Scheme

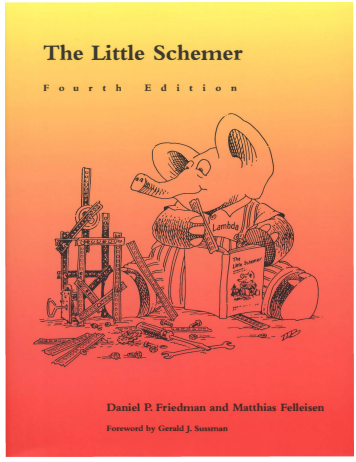
Orígenes

Scheme es un dialecto de Lisp. Scheme fue creado en el *MIT Computer Science and Artificial Intelligence Laboratory (MIT AI Lab)* por Guy L. Steele y Gerald Jay Sussman en 1975.

Introducción a Scheme

Algunos libros





Introducción a Scheme

Usando Scheme

(i) La implementación de Scheme usada en el curso es la de MIT/GNU la cual puede ser instalada con la siguiente instrucción:

```
$ sudo apt install mit-scheme
```

Introducción a Scheme

Usando Scheme

(i) La implementación de Scheme usada en el curso es la de MIT/GNU la cual puede ser instalada con la siguiente instrucción:

```
$ sudo apt install mit-scheme
```

(ii) Para ejecutar el interpretador de Scheme use la instrucción

```
$ mit-scheme
MIT/GNU Scheme running under GNU/Linux
...
Release 12.1 || SF || LIAR/x86-64
1 ]=>
```

(continua en la próxima diapositiva)

Introducción a Scheme

Usando Scheme (continuación)

(iii) Para evitar problemas con las teclas de flechas instale el programa `rlwrap` y use la instrucción

```
$ rlwrap mit-scheme
```

Usando Scheme (continuación)

(iii) Para evitar problemas con las teclas de flechas instale el programa `rlwrap` y use la instrucción

```
$ rlwrap mit-scheme
```

(iv) Para evitar digitar la anterior instrucción se puede adicionar la siguiente línea al archivo de inicialización de su *shell*:

```
alias mit-scheme='rlwrap mit-scheme'
```

Forma extendida de Backus-Naur (eBNF)

expresión $\rightarrow$ átomo | ‘(’ {expresión} ‘)’

átomo $\rightarrow$ número | cadena | símbolo | carácter | booleano

Forma extendida de Backus-Naur (eBNF)

$$\text{expresión} \rightarrow \text{átomo} \mid \text{`(' } \{\text{expresión}\} \text{ `)'}$$

$$\text{átomo} \rightarrow \text{número} \mid \text{cadena} \mid \text{símbolo} \mid \text{carácter} \mid \text{booleano}$$

Observación

La eBNF arriba es para una versión simplificada de Scheme.

Introducción a Scheme

Expresiones prefijas y con paréntesis

En Scheme las expresiones se escriben de forma prefija agrupadas con paréntesis.

*En la terminología de Scheme los términos «procedimiento» y «función» son sinónimos.

Introducción a Scheme

Expresiones prefijas y con paréntesis

En Scheme las expresiones se escriben de forma prefija agrupadas con paréntesis.

Ejemplo (expresión prefija)

```
(foo a b)
```

El procedimiento* `foo` es aplicado a los parámetros actuales `a` y `b`.

*En la terminología de Scheme los términos «procedimiento» y «función» son sinónimos.

Introducción a Scheme

Expresiones prefijas y con paréntesis

En Scheme las expresiones se escriben de forma prefija agrupadas con paréntesis.

Ejemplo (expresión prefija)

```
(foo a b)
```

El procedimiento* `foo` es aplicado a los parámetros actuales `a` y `b`.

Ejemplo (expresiones anidadas)

```
(foo (bar x) (baz y))
```

*En la terminología de Scheme los términos «procedimiento» y «función» son sinónimos.

Regla de evaluación

(i) Números, cadenas, caracteres y booleanos evalúan a si mismos.

Regla de evaluación

(i) Números, cadenas, caracteres y booleanos evalúan a si mismos.

(ii) Símbolos que no son palabras reservadas son considerados identificadores o variables. Éstos evalúan al valor que tienen en el actual entorno (formas especiales (*special forms*)).

Regla de evaluación

(i) Números, cadenas, caracteres y booleanos evalúan a si mismos.

(ii) Símbolos que no son palabras reservadas son considerados identificadores o variables. Éstos evalúan al valor que tienen en el actual entorno (formas especiales (*special forms*)).

(iii) Una expresión entre paréntesis (lista) es evaluada de la siguiente forma:

Regla de evaluación

(i) Números, cadenas, caracteres y booleanos evalúan a si mismos.

(ii) Símbolos que no son palabras reservadas son considerados identificadores o variables. Éstos evalúan al valor que tienen en el actual entorno (formas especiales (*special forms*)).

(iii) Una expresión entre paréntesis (lista) es evaluada de la siguiente forma:

  (a) Si la primera expresión es una palabra reservada entonces se aplica un regla especial para evaluar la expresión entre paréntesis.

Regla de evaluación

(i) Números, cadenas, caracteres y booleanos evalúan a si mismos.

(ii) Símbolos que no son palabras reservadas son considerados identificadores o variables. Éstos evalúan al valor que tienen en el actual entorno (formas especiales (*special forms*)).

(iii) Una expresión entre paréntesis (lista) es evaluada de la siguiente forma:

  (a) Si la primera expresión es una palabra reservada entonces se aplica un regla especial para evaluar la expresión entre paréntesis.

  (b) En otro caso, la expresión entre paréntesis representa la aplicación de una función. Cada expresión dentro los paréntesis es evaluada recursivamente evaluando primero los argumentos de la función (*applicative order evaluation*). La primera expresión debe evaluar a una función y el valor de las otras expresiones son los argumentos a la función.

Introducción a Scheme

Ejemplos (regla de evaluación)

expresión	representa	valor
`42`	número entero	`42`
`''hola''`	cadena	`''hola''`
`#\a`	el carácter 'a'	`#\a`
`#t`	el booleano «true»	`#t`
`#f`	el booleano «false»	`#f`
`a`	un símbolo	`a`
`(+ 2 3)`	una lista	`5`
`(* (+ 2 3) (/ 6 2))`	una lista	`15`

Introducción a Scheme

Convenciones

(i) El *prompt* del interpretador `mit-scheme` `n ]=>` será representado por `>`.

(ii) La notación `E` $\Rightarrow$ `v` denota que la expresión `E` es evaluada al valor `v`.

Datos y programas

Descripción

Una de las características más importante de Scheme es que los datos y los programas son representados de la misma forma.

Para evitar que el interpretador evalué una expresión como un procedimiento empleamos la forma especial `quote`.

Datos y programas

Ejemplo

```
> (1 2 3)
Error: The object 1 is not applicable.
```

Datos y programas

Ejemplo

```
> (1 2 3)
Error: The object 1 is not applicable.
```

```
> (quote (1 2 3))
⇒ (1 2 3)
```

Datos y programas

Ejemplo

```
> (1 2 3)
Error: The object 1 is not applicable.
```

```
> (quote (1 2 3))
⇒ (1 2 3)
```

```
> '(1 2 3)
⇒ (1 2 3)
```

Variables locales

Ejemplo (`let` y `letrec`)

En el tablero.

Procedimientos anónimos

Ejemplo (`lambda`)

En el tablero.

Definiendo variables y procedimientos

Descripción

En Scheme podemos definir variables usando la forma especial `define`.

Definiendo variables y procedimientos

Descripción

En Scheme podemos definir variables usando la forma especial `define`.

Ejemplo

```
(define my-var 5)
```

(i) La variable `my-var` es adicionada al contexto actual.

(ii) Scheme localiza memoria para la variable `my-var` y la variable es ligada (*bounded*) a dicha posición de memoria.

(iii) La variable es inicializada con el valor 5.

Definiendo variables y procedimientos

Descripción

En Scheme también podemos definir procedimientos usando la forma especial **`define`**.

Ejemplo

```
> (define (double x)
    (+ x x))
> (double 4)
⇒ 8
```

Chequeo de tipos dinámico

Características

- Las variables no tienen tipos pero los valores si.

Chequeo de tipos dinámico

Características

- Las variables no tienen tipos pero los valores si.
- El chequeo de tipos se realiza en tiempo de ejecución.

Chequeo de tipos dinámico

Características

- Las variables no tienen tipos pero los valores si.
- El chequeo de tipos se realiza en tiempo de ejecución.
- El chequeo de tipos dinámico es costoso en general pero algunas implementaciones reducen este costo haciendo inferencia de tipos en tiempo de compilación.

Chequeo de tipos dinámico

Ejemplo

```
> (define square (lambda (n) (* n n)))
> (square "hello")
Error: The object "hello", passed as the first argument to integer-zero?,
is not the correct type.
```

Chequeo de tipos dinámico

Pregunta

¿*Dynamically typed* = *untyped*?

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

Ejemplo (función factorial)

Mirar el archivo `fp/tail-recursion/factorial/scheme/factorial`.

```
$ cd fp/tail-recursion/factorial/scheme
$ rlwrap mit-scheme
```

Recursión de cola

Ejemplo (función factorial)

Mirar el archivo `fp/tail-recursion/factorial/scheme/factorial`.

```
$ cd fp/tail-recursion/factorial/scheme
$ rlwrap mit-scheme
```

```
> (load "factorial")

> (fact-ntr 6)
⇒ 720

> (fact-tr 6 1)
⇒ 720
```

Listas

Descripción

(i) Las listas son objetos compuestos en Scheme.

(ii) Las listas están formadas por una cabeza y una cola.

(iii) La lista vacía se denota por `()`.

(iv) Constructor: Dado un elemento `x` y una lista `xs`, el valor de la expresión `(cons x xs)` es una nueva lista con cabeza `x` y cola `xs`.

(v) Dos selectores: El valor de la expresión `(car xs)` es la cabeza de la lista `xs` y el valor de la expresión `(cdr xs)` es la cola de lista `xs`.

(vi) Predicado: El valor de la expresión `(null? xs)` es `#t` si `xs` es la lista vacía, de lo contrario su valor es `#f`.

Listas

Descripción

(i) Las listas son objetos compuestos en Scheme.

(ii) Las listas están formadas por una cabeza y una cola.

(iii) La lista vacía se denota por `()`.

(iv) Constructor: Dado un elemento `x` y una lista `xs`, el valor de la expresión `(cons x xs)` es una nueva lista con cabeza `x` y cola `xs`.

(v) Dos selectores: El valor de la expresión `(car xs)` es la cabeza de la lista `xs` y el valor de la expresión `(cdr xs)` es la cola de lista `xs`.

(vi) Predicado: El valor de la expresión `(null? xs)` es `#t` si `xs` es la lista vacía, de lo contrario su valor es `#f`.

Observación

Las funciones `cons`, `car`, `cdr` y `null?` son *built-in*.

Listas

Ejemplo (definición de concatenación de listas)

```
1 (define (my-append l m)
2   (if (null? l)
3     m
4     (cons (car l) (my-append (cdr l) m))))
```

Listas

Otros procedimientos sobre listas

Otros procedimientos *built-in* sobre listas son: `length`, `list`, `append`, `reverse` y `member`.

Ejemplos

En el tablero.

Funciones de orden superior

Definición

Una función es de **orden superior** (*higher-order*) 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 (`comp`)

(i) La función (el operador) de composición `comp` compone dos funciones, es decir,

`(comp g f) x` $\Rightarrow$ `g (f x)`.

Funciones de orden superior

Ejemplo (`comp`)

(i) La función (el operador) de composición `comp` compone dos funciones, es decir,

`(comp g f) x` $\Rightarrow$ `g (f x)`.

(ii) Implementación:

```
(define (comp g f)
  (lambda (x) (g (f x))))
```

Mirar el archivo `fp/higher-order/higher-order.scm`.

Funciones de orden superior

Ejemplo (`map`)

(i) La función `map` aplica una función a cada elemento de una lista, es decir,

`(map f '(x1 x2 ... xn))` $\Rightarrow$ `((f x1) (f x2) ... (f xn))`

Funciones de orden superior

Ejemplo (`map`)

(i) La función `map` aplica una función a cada elemento de una lista, es decir,

```
(map f '(x1 x2 ... xn)) ⇒ ((f x1) (f x2) ... (f xn))
```

(ii) Leyes:

```
(map f '())          ⇒  '()
(map f (cons x xs))  ⇒  (cons (f x) (map f xs))
```

Funciones de orden superior

Ejemplo (`map`)

(i) La función `map` aplica una función a cada elemento de una lista, es decir,

`(map f '(x1 x2 ... xn))` $\Rightarrow$ `((f x1) (f x2) ... (f xn))`

(ii) Leyes:

`(map f '())` $\Rightarrow$ `'()`

`(map f (cons x xs))` $\Rightarrow$ `(cons (f x) (map f xs))`

(iii) Implementación:

```
1 (define (my-map f xs)
2   (if (null? xs)
3       '()
4        (cons (f (car xs)) (my-map f (cdr xs)))))
```

Mirar el archivo `fp/higher-order/higher-order.scm`.

Funciones de orden superior

Ejemplo (`filter`)

(i) La función `filter` selecciona los elementos de una lista que satisfacen una propiedad.

Funciones de orden superior

Ejemplo (`filter`)

(i) La función `filter` selecciona los elementos de una lista que satisfacen una propiedad.

(ii) Leyes:

`(filter p? '())`	$\Rightarrow$	`'()`
`(filter p? (cons x xs))`	$\Rightarrow$	`(cons x (filter p? xs))`, si `(p? x)`
`(filter p? (cons x xs))`	$\Rightarrow$	`(filter p? xs)`, si (not (p? x))

Funciones de orden superior

Ejemplo (`filter`)

(i) La función `filter` selecciona los elementos de una lista que satisfacen una propiedad.

(ii) Leyes:

`(filter p? '())` $\Rightarrow$ `'()`

`(filter p? (cons x xs))` $\Rightarrow$ `(cons x (filter p? xs))` , si `(p? x)`

`(filter p? (cons x xs))` $\Rightarrow$ `(filter p? xs)` , si (not (p? x))

(iii) Implementación:

```
1 (define (my-filter p? xs)
2 (if (null? xs)
3     '()
4     (if (p? (car xs))
5         (cons (car xs) (filter p? (cdr xs)))
6         (filter p? (cdr xs)))))
```

Funciones de orden superior

Ejemplo (`foldr`)

(i) La función `foldr` combina los elementos de una lista, de derecha a izquierda, usando una función binaria y un valor inicial, es decir

`(foldr f z '(x1 x2 ... xn))` $\Rightarrow$ `(f x1 (f x2 ... (f xn z)))`

Funciones de orden superior

Ejemplo (`foldr`)

(i) La función `foldr` combina los elementos de una lista, de derecha a izquierda, usando una función binaria y un valor inicial, es decir

`(foldr f z '(x1 x2 ... xn))` $\Rightarrow$ `(f x1 (f x2 ... (f xn z)))`

(ii) Leyes:

`(foldr f z '())` $\Rightarrow$ `z`

`foldr f z (cons x xs))` $\Rightarrow$ `(f x (foldr f z xs))`

Funciones de orden superior

Ejemplo (`foldr`)

(i) La función `foldr` combina los elementos de una lista, de derecha a izquierda, usando una función binaria y un valor inicial, es decir

`(foldr f z '(x1 x2 ... xn))` $\Rightarrow$ `(f x1 (f x2 ... (f xn z)))`

(ii) Leyes:

`(foldr f z '())` $\Rightarrow$ `z`

`foldr f z (cons x xs))` $\Rightarrow$ `(f x (foldr f z xs))`

(iii) Implementación:

```
1 (define (foldr f z xs)
2   (if (null? xs)
3       z
4       (f (car xs) (foldr f z (cdr xs)))))
```

Mirar el archivo `fp/higher-order/higher-order.scm`.

Tema

Introducción a Haskell

Características importantes

- Haskell es un lenguaje de programación funcional pura con evaluación perezosa (*lazy*).
- Haskell, al igual que todos los lenguajes funcionales, soporta las 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 chequeo de tipos polimórficos *polymorphic type checking*.

  Discusión: *«Well-type programs cannot "go wrong"»* [Milner 1978, pág. 348].

(continua en la próxima diapositiva)

Introducción a Haskell

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

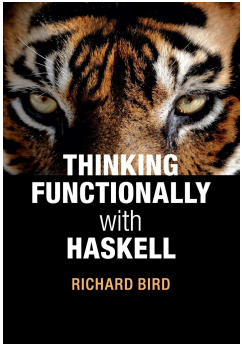
El artículo fue escrito en 1984 y circuló informalmente. El artículo no usó Haskell sino Miranda, un lenguaje predecesor de Haskell.

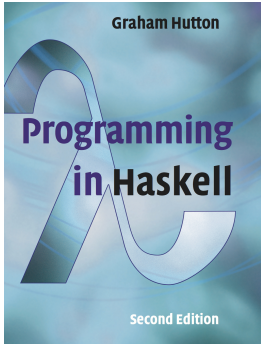
Introducción a Haskell

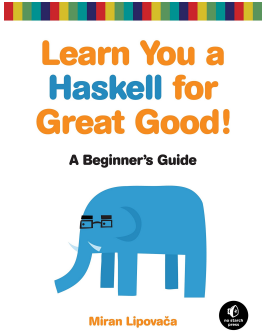
Orígenes

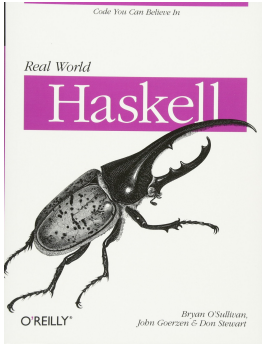
Haskell fue definido por un comité. El *«Haskell version 1.0 Report»* fue publicado el primero de abril de 1990. Una historia muy completa es presentada en [Hudak, Hughes, Peyton Jones y Wadler 2007].

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]
```

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

En Haskell las listas son *built-in*, donde una lista es:

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

El operador binario `:` 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 length1 :: [a] -> Int
2 length1 []       = 0
3 length1 (x : xs) = 1 + length1 xs
```

Polimorfismo paramétrico

Ejemplo

Concatenación de dos listas.

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

Caracteres y cadenas

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

Ejemplo

```
1 'a'                    -- Character.
2 'a' : 'b' : 'c' : []   -- List of characters.
3 ['a','b','c']          -- List of characters.
4 "abc"                  -- String.
5
6 -- List of strings.
7 ["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 (evaluación de cortocircuito (*short-cort evaluation*))

```haskell
-- Disjunction
(||) :: Bool -> Bool -> Bool

foo :: Int -> Bool  -- Non-terminating funcion
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n
```

Evaluación perezosa

Ejemplo (evaluación de cortocircuito (*short-cort evaluation*))

```haskell
-- Disjunction
(||) :: Bool -> Bool -> Bool

foo :: Int -> Bool  -- Non-terminating funcion
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n
```

Pregunta

¿Cuál es el valor de `bar 10`?

Evaluación perezosa

Ejemplo (evaluación de cortocircuito (*short-cort evaluation*))

```haskell
-- Disjunction
(||) :: Bool -> Bool -> Bool

foo :: Int -> Bool  -- Non-terminating funcion
foo n = foo (n + 1)

bar :: Int -> Bool
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`.

```
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`.

```
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`.

```
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
```

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

Tipos de datos algebraicos

Ejemplo

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

donde `True` y `False` son los (data) constructores del tipo de dato `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) + fib (n - 2)
```

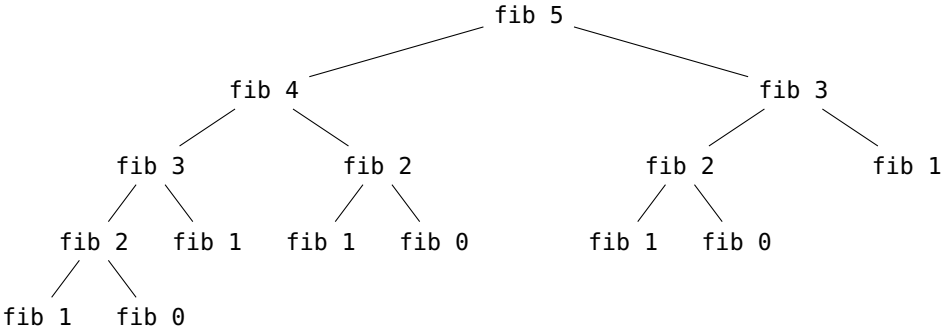
El número de llamadas a `fib` crece exponencialmente con el tamaño de `n`.

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

(continua en la próxima diapositiva)

Eficiencia de la recursión

Ejemplo (función de Fibonnaci)



(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
```

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

(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
```

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

```
$ make fib2
$ time ./fib2
real      0m0.006s
```

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

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

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

```
1 reverse :: [a] -> [a]
2 reverse xs = rev xs []
3   where
4   rev []       zs = zs
5   rev (y : ys) zs = rev ys (y : zs)
```

Mirar el archivo `fp/Reverse.hs`.

Recursión de cola

Observación

En las optimizaciones de recursión de cola en Haskell, los argumentos de acumulación deben ser estrictos, es decir, se deben usar *bang patterns*.

Recursión de cola

Ejemplo (función factorial)

Tiempos de ejecución para los programas en el directorio `fp/tail-recursion/factorial/haskell`.

(i) No recursión de cola ni *bang patterns*

```
$ make fact1
$ time ./fact1
...
real     0m19,719s
```

(ii) No recursión de cola pero si *bang patterns*

```
$ make fact2
$ time ./fact2
...
real     0m19,333s
```

(iii) Recursión de cola pero no *bang patterns*

```
$ make fact3
$ time ./fact3
...
real     0m20,273s
```

(iv) Recursión de cola y *bang patterns*

```
$ make fact4
$ time ./fact4
...
real     0m3,788s
```

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.

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

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
(.) g f = \ x -> g (f x)
```

Mirar el archivo `fp/higher-order-functions/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 f []       = []
3 map f (x : xs) = f x : map f xs
```

Mirar el archivo `fp/higher-order-functions/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/higher-order-functions/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 función `a -> Bool`).

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

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter p []       = []
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 (versión simple):

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

Type Classes

Ejemplo

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

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = 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
```

Entrada y salida

Un problema

¿Cómo pueden programas con entrada y salida ser escritos usando solamente funciones puras?

Entrada y salida

Un problema

¿Cómo pueden programas con entrada y salida ser escritos usando solamente funciones puras?

Una solución

Hay varias formas de usar funciones puras y efectos colaterales (véase, *p. ej.* [Peyton Jones y Wadler 1993]). La solución en Haskell es vía *mónadas*.

Entrada y salida

Definición

El **tipo unidad (*unit type*)** es un tipo con un solo elemento. En Haskell, el tipo unidad y su elemento son

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

El tipo unidad es usado cuando se realiza entrada y salida.

Entrada y salida

The `IO` type

La siguiente descripción del tipo `IO` está en [Hutton 2016, § 10.2].

Entrada y salida

The `IO` type

La siguiente descripción del tipo `IO` está en [Hutton 2016, § 10.2].

Un programa con entrada y salida puede ser representado por una función

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

Entrada y salida

The `IO` type

La siguiente descripción del tipo `IO` está en [Hutton 2016, § 10.2].

Un programa con entrada y salida puede ser representado por una función

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

¿Y si el programa además retorna un valor?

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

(continua en la próxima diapositiva)

Entrada y salida

The `IO` type (continuación)

¿Y si el programa además requiere un argumento?

Por ejemplo, el tipo de un programa requiriendo un carácter y retornando un entero es

```
Char -> IO Int
```

es decir,

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

Entrada y salida

The `IO` type (continuación)

¿Y si el programa además requiere un argumento?

Por ejemplo, el tipo de un programa requiriendo un carácter y retornando un entero es

```
Char -> IO Int
```

es decir,

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

Es responsabilidad del compilador manejar el estado del `World`. El tipo `IO a` es un tipo primitivo en Haskell.

Entrada y salida

Definición

Una **acción (*action*)** es un término (o expresión) de tipo `IO a`. Cuando el término es evaluado la acción es llevada a cabo.

Entrada y salida

Definición

Una **acción (*action*)** es un término (o expresión) de tipo `IO a`. Cuando el término es evaluado la acción es llevada a cabo.

Ejemplo

- `t : IO Char` es una acción que retorna un carácter.
- `t : IO ()` es una acción que no retorna un valor, donde `()` es el tipo unidad.

Entrada y salida

Descripción

El tipo de dato abstracto (*abstract datatype*) `IO a` tiene (al menos) las siguientes operaciones [Bird 1998, § 10.1]:

```
1 return  :: a -> IO a
2 (>>=)   :: IO a -> (a -> IO b) -> IO b
3 putChar :: Char -> IO ()
4 getChar :: IO Char
```

Entrada y salida

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`

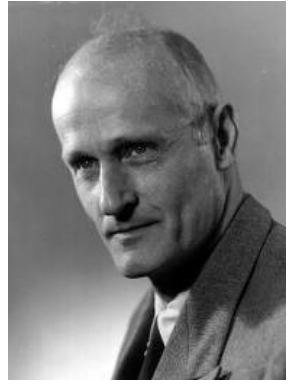
Demo

En clase.

Tema

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.
- Base de los lenguajes de programación funcional.
- 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.

Cálculo lambda

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:

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

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,

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

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

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

Applicative order reduction

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

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

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

Cálculo lambda

Observación

Church [1935, 1936] demostró que el conjunto

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

es indecidible. Éste fue el primer conjunto indecible.

Cálculo lambda

Descripción

`LambdaShell` es un REPL (*Read Evaluate Print Loop*) implementado en Haskell para el cálculo lambda.*

*`https://hackage.haskell.org/package/LambdaShell`.

Cálculo lambda

Descripción

`LambdaShell` es un REPL (*Read Evaluate Print Loop*) implementado en Haskell para el cálculo lambda.*

Instalación y uso

Para instalar (se requiere `GHC >= 9.10.1`) y usar `LambdaShell` siga las siguientes instrucciones:

```
$ git clone https://github.com/asr/lambda-shell
$ cd lambda-shell
$ cabal install
$ lambdaShell

$ >: help
```

Véase además el archivo `Prelude.lam` en el directorio del `LambdaShell`.

*`https://hackage.haskell.org/package/LambdaShell`.

Tema

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. 171, 172).


Bird, Richard [1988] (1998). Introduction to Functional Programming. 2.ª ed. Prentice Hall Press (vid. pág. 150).


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. 180).


— (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, págs. 345-363. DOI: 10.2307/2371045 (vid. pág. 180).


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. 157, 158).


Hindley, J. Roger y Seldin, Jonathan P. (2008). Lambda-Calculus and Combinators. An Introduction. Cambridge University Press (vid. págs. 171, 172).

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. 75).


Hudak, Paul, Peterson, John y Fasel, Joseph H. (1999). A Gentle Introduction to Haskell 98. URL: https://www.haskell.org/tutorial/ (vid. págs. 113-115).


Hughes, J. (1989). Why Functional Programming Matters. The Computer Journal 32.2, págs. 98-107 DOI: 10.1093/comjnl/32.2.98 (vid. págs. 73, 74).


Hutton, Graham [2007] (2016). Programming in Haskell. 2.ª ed. Cambridge University Press (vid. págs. 9, 10, 143-145).


Louden, Kenneth C. y Lambert, Kenneth A. [1993] (2011). Programming Languages. Principles and Practice. 3.ª ed. Cengage Learning (vid. pág. 2).


Milner, Robin (1978). A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences 17.3, págs. 348-375. DOI: 10.1016/0022-0000(78)90014-4 (vid. pág. 71).


O'Sullivan, Bryan, Goerzen, John y Stewart, Don (2008). Real World Haskell. O'Really Media, Inc. (vid. págs. 9, 10).

Referencias

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. 140, 141).