Funciones de Orden Superior

Ernesto Rodriguez

Universidad del Itsmo

erodriguez@unis.edu.gt

Funciones de Orden Superior

- Los lenguajes funcionales no diferencian funciones de valores
- Esto permite escribir funciones que aceptan otras funciones como parametro
- Este patron permite reemplazar muchos componentes tradicionales (ciclos, manejo de excepciones, saltos, etc.) mediante funciones.
  - Los Lambda Papers[2] describen diferentes formas de alcanzar esto.
- La mayoria de lenguajes modernos adopta este patron.
  - **Java 8** por fin llego a donde Church estaba en 1930. – Phillip Wadler
- Un patron llamado **Programación Funcional Reactiva** permite crear interfaces graficas mediante **funciones de orden superior**. Este Patron se utiliza en Elm, React y muchas otras herramientas.

Ejemplo: Composición

- ¿Que **tipo** deberia tener la composición?
- ¿Se puede programar la composición en Elm?

Ciclos

- Consideremos las siguientes funciones sobre listas:
  - `duplicar` : $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$
  - `negar` : $\mathcal{L}(\mathbb{B}) \to \mathcal{L}(\mathbb{B})$
- ¿Tiene algun problema la implementación?
- ¿Podemos simplificar?

Ciclos

- Consideremos las siguientes funciones sobre listas:
  - `duplicar` : $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$
  - `negar` : $\mathcal{L}(\mathbb{B}) \to \mathcal{L}(\mathbb{B})$
- ¿Tiene algun problema la implementación?
  - Repetición de codigo
- ¿Podemos simplificar?
  - Idea: Abstraer la recursión

Ciclos

La función `map` : $(a \to b) \to \mathcal{L}(a) \to \mathcal{L}(b)$ hace exactamente eso:

Ciclos

La función `map` : $(a \to b) \to \mathcal{L}(a) \to \mathcal{L}(b)$ hace exactamente eso:

$$\texttt{map}\ \textit{lista}\ f = \begin{cases} [\,] & \text{if } \textit{lista} \equiv [\,] \\ \texttt{f x} :: \texttt{map xs f} & \text{if } \textit{lista} \equiv x :: xs \end{cases}$$

¿Como implementamos `duplicar` y `negar` utilizando `map`?

Ciclos

La función `map` : $(a \to b) \to \mathcal{L}(a) \to \mathcal{L}(b)$ hace exactamente eso:

$$\texttt{map}\ f\ \mathit{lista} = \begin{cases} [\,] & \text{if } \mathit{lista} \equiv [\,] \\ \texttt{f x} :: \texttt{map f xs} & \text{if } \mathit{lista} \equiv x :: xs \end{cases}$$

¿Como implementamos `duplicar` y `negar` utilizando `map`?

- `duplicar` = `map` $((*)\ 2)$
- `negar` = `map not`

Funciones Anonimas y Let

- A menudo se definen funciónes de uso unico.
- Es conveniente definir la función en donde sera utilizada
- A veces se necesita contexto local para definir la función.
- Ejemplo: `multiplicar` : $\mathbb{Z} \to \mathcal{L}(\mathtt{Z}) \to \mathcal{L}(\mathtt{Z})$

Funciones Anonimas y Let

- A menudo se definen funciónes de uso unico.
- Es conveniente definir la función en donde sera utilizada
- A veces se necesita contexto local para definir la función.
- Ejemplo: `multiplicar` : $\mathbb{Z} \to \mathcal{L}(\mathtt{Z}) \to \mathcal{L}(\mathtt{Z})$

```
multiplicar factor lista =
  let
    op valor = factor * valor
  in
    map op lista
```

```
multiplicar factor lista =
  map (λvalor → factor * valor) lista
```

Generalizando los ciclos de Listas

- ¿Existen funciones que no se pueden expresar mediante `map`?

Generalizando los ciclos de Listas

- ¿Existen funciones que no se pueden expresar mediante `map`?
  - ¿`count` : $\mathcal{L}(a) \to \mathbb{N}$?
  - ¿`sumatoria` : $\mathcal{L}(\mathbb{Z}) \to \mathbb{Z}$?

Generalizando los ciclos de Listas

- ¿Existen funciones que no se pueden expresar mediante `map`?
  - ¿`count` : $\mathcal{L}(a) \to \mathbb{N}$?
  - ¿`sumatoria` : $\mathcal{L}(\mathbb{Z}) \to \mathbb{Z}$?
- El tipo de la función esta incorrecto, ya que `map` solo puede producir listas.

Primer intento

Llamaremos a la nueva función `fold` : $(a \to b) \to \mathcal{L}(a) \to b$:

$$\texttt{fold}\ f\ \mathit{lista} = \begin{cases} ? & \text{if } \mathit{lista} \equiv [\,] \\ f\ x & \text{if } \mathit{lista} \equiv x :: xs \end{cases}$$

- **Problema:** No es posible retornar un valor cuando *lista* esta vacia.

Primer intento

Llamaremos a la nueva función `fold` : $(a \to b) \to b \to \mathcal{L}(a) \to b$:

$$\texttt{fold}\ f\ base\ lista = \begin{cases} base & \text{if } lista \equiv [\,] \\ f\ x & \text{if } lista \equiv x :: xs \end{cases}$$

- **Problema:** No es posible retornar un valor cuando *lista* esta vacia.
  - Aceptamos un valor de retorno para cubrir ese caso

Segundo intento

Llamaremos a la nueva función `fold` : $(a \to b) \to b \to \mathcal{L}(a) \to b$:

$$\texttt{fold}\ f\ base\ lista = \begin{cases} base & \text{if } lista \equiv [\,] \\ f\ x & \text{if } lista \equiv x :: xs \end{cases}$$

- **Problema:** No es posible retornar un valor cuando *lista* esta vacia.
  - Aceptamos un valor de retorno para cubrir ese caso
- **Problema:** El valor de retorno solo depende del primer valor.

Segundo intento

Idea:

- "Recordar" el valor producido por cada elemento de la lista
- Permitir que la **funcion de orden superior** o el **reductor** utilize ese valor para producir un nuevo valor
- Retornar el ultimo valor que haya sido calculado.

Segundo intento

Llamaremos a la nueva función `fold` : $(b \to a \to b) \to b \to \mathcal{L}(a) \to b$:

$$\texttt{fold}\ f\ base\ lista = \begin{cases} base & \text{if } lista \equiv [\,] \\ f\ (\texttt{fold}\ f\ base\ xs)\ x & \text{if } lista \equiv x :: xs \end{cases}$$

- **Problema:** No es posible retornar un valor cuando *lista* esta vacia.
  - Aceptamos un valor de retorno para cubrir ese caso
- **Problema:** El valor de retorno solo depende del primer valor.
  - Recordar el valor que fue calculado en el paso anterior

Fold: Recursión Generalizada

- ¿Podemos implementar la función `count`?
- ¿Podemos implementar la función `sumatoria`?
- ¿Podemos implementar la función `map`?

Fold: Recursión Generalizada

- ¿Podemos implementar la función `count`?
  - `sumatoria` $xs = \texttt{fold}\ (\lambda\ a\ \_ \rightarrow a + 1)\ 0\ xs$
- ¿Podemos implementar la función `sumatoria`?
- ¿Podemos implementar la función `map`?

Fold: Recursión Generalizada

- ¿Podemos implementar la función `count`?
  - `sumatoria` $xs = $ `fold` $(\lambda\ a\ \_ \rightarrow a + 1)\ 0\ xs$
- ¿Podemos implementar la función `sumatoria`?
  - `sumatoria` $xs = $ `fold` $(\lambda\ a\ b \rightarrow a + b)\ 0\ xs$
- ¿Podemos implementar la función `map`?

Fold: Recursión Generalizada

- ¿Podemos implementar la función `count`?
  - `sumatoria` $xs = $ `fold` $(\lambda\ a\ \_ \rightarrow a + 1)\ 0\ xs$
- ¿Podemos implementar la función `sumatoria`?
  - `sumatoria` $xs = $ `fold` $(\lambda\ a\ b \rightarrow a + b)\ 0\ xs$
- ¿Podemos implementar la función `map`?
  - `map` $f\ as = $ `fold` $(\lambda\ bs\ a \rightarrow f\ a :: bs)\ [\,]\ as$
- La función `fold` generaliza todas las **funciones recursivas** sobre listas. Ver "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" [1].
- La generalización de la función `fold` para cualquier tipo se conoce como un **catamorfismo**.

Referencias

Erik Meijer, Maarten Fokkinga, and Ross Paterson.
Functional programming with bananas, lenses, envelopes and barbed wire.
pages 124–144. Springer-Verlag, 1991.

Guy L. Steele and Gerald Jay Sussman.
Lambda papers.
https://en.wikisource.org/wiki/Lambda_Papers.