Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Parsers aplicativos

Combinadores de parsing

Los combinadores de parsing forman un EDSL que es implementado usando un shallow embedding.

Están formados por dos grupos de funciones:

- Funciones básicas que sirven para reconocer determinados strings de entrada
- Un grupo de combinadores que permiten construir nuevos parsers a partir de otros.

Parsers elementales

La mayoría de las bibliotecas de parsing están formadas por los siguientes 4 combinadores básicos:

string vacío	*pSucceed*
terminales	*pSym s*
alternativa	*p* <|> *q*
composición	*p* <*> *q*

El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo a:

$$String \rightarrow a$$

El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo a:

$$String \rightarrow a$$

- Un parser podría ser ambiguo y retornar varios posibles valores, significando que puede haber varias formas de reconocer la entrada.

$$String \rightarrow [a]$$

El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo a:

$$String \rightarrow a$$

- Un parser podría ser ambiguo y retornar varios posibles valores, significando que puede haber varias formas de reconocer la entrada.

$$String \rightarrow [a]$$

- Un parser podría no consumir toda la entrada y retornar además la parte de la entrada no consumida.

$$String \rightarrow [(a, String)]$$

El tipo de un parser

En resumen,

$$\texttt{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

El tipo de un parser

En resumen,

$$\texttt{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

Podemos abstraer el tipo *String*:

$$\texttt{type}\ \mathit{Parser}\ s\ a = \mathit{Eq}\ s \Rightarrow [s] \to [(a, [s])]$$

- En su lugar ponemos una lista de valores de tipo s.
- A los valores de tipo s les vamos a requerir que sean comparables por igualdad (instancia de la clase *Eq*).

Combinadores básicos

$$
\begin{aligned}
\mathit{pFail} &:: && && \mathit{Parser}\ s\ a \\
\mathit{pSucceed} &:: a && && \to \mathit{Parser}\ s\ a \\
\mathit{pSym} &:: \mathit{Eq}\ s \Rightarrow s && && \to \mathit{Parser}\ s\ s \\
\texttt{<|>} &:: \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ a \\
\texttt{<*>} &:: \mathit{Parser}\ s\ (a \to b) && \to \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ b
\end{aligned}
$$

Combinadores básicos

$$pFail :: Parser\ s\ a$$
$$pFail = \lambda cs \rightarrow [\,]$$

Combinadores básicos

$$pFail :: Parser\ s\ a$$
$$pFail = \lambda cs \rightarrow [\,]$$

$$pSucceed :: a \rightarrow Parser\ s\ a$$
$$pSucceed\ a = \lambda cs \rightarrow [(a, cs)]$$

Combinadores básicos

```
pFail :: Parser s a
pFail = λcs → []
```

```
pSucceed    :: a → Parser s a
pSucceed a = λcs → [(a, cs)]
```

```
pSym    :: Eq s ⇒ s → Parser s s
pSym s = λcs → case cs of
                 []       → []
                 (c : cs′) → if c == s
                               then [(c, cs′)]
                               else []
```

Combinadores básicos

$$(<|>) \quad :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a$$
$$p\ <|>\ q = \lambda cs \to p\ cs +\!\!+ q\ cs$$

Combinadores básicos

$$
\begin{aligned}
&(<|>) \quad :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
&p\ <|>\ q = \lambda cs \to p\ cs +\!\!+ q\ cs
\end{aligned}
$$

$$
\begin{aligned}
&(<*>) \quad :: \mathit{Parser}\ s\ (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \\
&(p <*> q)\ cs = [(f\ a, cs'') \mid (f, cs') \leftarrow p\ cs \\
&\qquad\qquad\qquad\qquad\qquad\qquad , (a, cs'') \leftarrow q\ cs']
\end{aligned}
$$

Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

- Reconocer una 'A', seguida de una 'B', y retornar ambos caracteres en un par.

Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

- Reconocer una 'A', seguida de una 'B', y retornar ambos caracteres en un par.

$$pAB = pSucceed\ (,)\ \texttt{<*>}\ pSym\ \texttt{'A'}\ \texttt{<*>}\ pSym\ \texttt{'B'}$$

Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo a. Toma como parámetro un parser que retorna un a.

Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo a. Toma como parámetro un parser que retorna un a.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:)\ \texttt{<*>}\ p\ \texttt{<*>}\ pList\ p \\
&\qquad\qquad \texttt{<|>} \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo a. Toma como parámetro un parser que retorna un a.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:)\ \texttt{<*>}\ p\ \texttt{<*>}\ pList\ p \\
&\qquad\qquad \texttt{<|>} \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

- Parser que reconoce un string de la forma $(AB)^*$.

Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo a. Toma como parámetro un parser que retorna un a.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:)\ \texttt{<*>}\ p\ \texttt{<*>}\ pList\ p \\
&\qquad\qquad \texttt{<|>} \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

- Parser que reconoce un string de la forma $(AB)^*$.

$$pListAB = pList\ pAB$$

Otros combinadores útiles

```
(<$>)  :: (a → b) → Parser s a → Parser s b
f <$> p = pSucceed f <*> p
```

```
opt       :: Parser s a → a → Parser s a
p `opt` a = p <|> pSucceed a
```

```
pSat   :: (s → Bool) → Parser s s
pSat p = λcs → case cs of
                 []       → []
                 (c : cs') → if p c
                               then [(c, cs')]
                               else []
```

Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \texttt{<\$>} pSym\ \texttt{'A'} \texttt{<*>} pSym\ \texttt{'B'}$$

Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \texttt{<\$>} pSym\ \texttt{'A'} \texttt{<*>} pSym\ \texttt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \texttt{<\$>} pSym\ \texttt{'A'} \texttt{<*>} pSym\ \texttt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

- Reconocer un dígito:

$$pDigit = pSat\ isDigit$$
$$\texttt{where}$$
$$isDigit\ c = (c \geqslant \texttt{'0'}) \wedge (c \leqslant \texttt{'9'})$$

Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \texttt{<\$>} pSym\ \texttt{'A'} \texttt{<*>} pSym\ \texttt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

- Reconocer un dígito:

$$pDigit = pSat\ isDigit$$
$$\texttt{where}$$
$$isDigit\ c = (c \geqslant \texttt{'0'}) \wedge (c \leqslant \texttt{'9'})$$

- Definición de *pList* usando <$> y *opt*:

$$pList\ p = (:) \texttt{<\$>} p \texttt{<*>} pList\ p\ `opt`\ []$$

Selección de resultados de parsers

```
(<*)  :: Parser s a → Parser s b → Parser s a
p <* q = (λx _ → x) <$> p <*> q
```

```
(*>)  :: Parser s a → Parser s b → Parser s b
p *> q = (λ_ y → y) <$> p <*> q
```

```
(<$)  :: a → Parser s b → Parser s a
a <$ q = pSucceed a <* q
```

Selección de resultados de parsers

$$(<*) \quad :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a$$
$$p <* q = (\lambda x\ \_ \to x) <\$> p <*> q$$

$$(*>) \quad :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ b$$
$$p *> q = (\lambda \_\ y \to y) <\$> p <*> q$$

$$(<\$) \quad :: a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a$$
$$a <\$ q = \mathit{pSucceed}\ a <* q$$

Ejemplo. Reconocer algo entre paréntesis.

$$\mathit{pParens}\ p = \mathit{pSym}\ '(' *> p <* \mathit{pSym}\ ')'$$

Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Functores Aplicativos

Functor

Un **functor** puede entenderse como un constructor de tipo $f :: * \to *$ junto a una función de tipo

$$(a \to b) \to f\ a \to f\ b$$

que permite mapear/reemplazar los valores de tipo a contenidos en una estructura de tipo $f\ a$ por valores de tipo b.

Functor

En Haskell el concepto de functor es capturado por una clase:

```
class Functor (f :: * → *) where
  fmap :: (a → b) → f a → f b
```

Functor

En Haskell el concepto de functor es capturado por una clase:

$$\texttt{class } \mathit{Functor}\ (f :: * \to *)\ \texttt{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

Para ser efectivamente un functor la función *fmap* debe satisfacer las siguientes propiedades:

$$\mathit{fmap}\ \mathit{id} = \mathit{id}$$
$$\mathit{fmap}\ (f.g) = \mathit{fmap}\ f.\mathit{fmap}\ g$$

que deberian ser chequeadas al definir cada instancia de la clase.

Ejemplos

```
instance Functor [] where
  fmap = map
```

Ejemplos

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

Ejemplos

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)

instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x)  = Left x
```

Ejemplos

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)

instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x)  = Left x

instance Functor ((->) r) where
  fmap f h = λr -> f (h r)   -- o sea, f.h
```

Modelando Error con `Maybe`

División segura

$$
\begin{aligned}
\mathit{divM}\ x\ y \mid y \not\equiv 0 \quad &= \mathit{Just}\ (x\ `\mathit{div}`\ y) \\
\mid \mathit{otherwise} &= \mathit{Nothing}
\end{aligned}
$$

Modelando Error con `Maybe`

División segura

$$\begin{aligned} \mathit{divM}\ x\ y \mid y \not\equiv 0 \quad &= \mathit{Just}\ (x\ `\mathit{div}`\ y) \\ \mid \mathit{otherwise} &= \mathit{Nothing} \end{aligned}$$

Con *fmap* puede aplicar una función pura al resultado de una división

$$\mathit{foo}\ x\ y = \mathit{fmap}\ (+2)\ (\mathit{divM}\ x\ y)$$

Modelando Error con `Maybe`

División segura

$$\begin{aligned} \mathit{divM}\ x\ y \mid y \not\equiv 0 \quad &= \mathit{Just}\ (x\ `\mathit{div}`\ y) \\ \mid \mathit{otherwise} &= \mathit{Nothing} \end{aligned}$$

Con *fmap* puede aplicar una función pura al resultado de una división

$$\mathit{foo}\ x\ y = \mathit{fmap}\ (+2)\ (\mathit{divM}\ x\ y)$$

en lugar de hacer:

$$\begin{aligned} \mathit{foo}\ x\ y = \texttt{case}\ \mathit{divM}\ x\ y\ \texttt{of} \\ \mathit{Just}\ r \quad &\to \mathit{Just}\ (r+2) \\ \mathit{Nothing} &\to \mathit{Nothing} \end{aligned}$$

Functores Aplicativos

Los **functores aplicativos** son functores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

Functores Aplicativos

Los **functores aplicativos** son functores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

Se debe cumplir que:

```
fmap f x = pure f <*> x
```

Sinónimo en *Applicative*:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
f <$> t = fmap f t
```

Ejemplo: Maybe

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

Ejemplo: Maybe

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

Puedo por ejemplo modelar expresiones con errores:

```
type Expr = Maybe Int

valE x   = pure x
addE x y = (+) <$> x <*> y
divE x y = case (x, y) of
             (Just vx, Just vy) → divM vx vy
             _                  → Nothing
```

Leyes de functores aplicativos

Identidad:

$$pure\ id\ \texttt{<*>}\ u \equiv u$$

Composición:

$$pure\ (.)\ \texttt{<*>}\ u\ \texttt{<*>}\ v\ \texttt{<*>}\ w \equiv u\ \texttt{<*>}\ (v\ \texttt{<*>}\ w)$$

Homomorfismo:

$$pure\ f\ \texttt{<*>}\ pure\ x \equiv pure\ (f\ x)$$

Intercambio:

$$u\ \texttt{<*>}\ pure\ x \equiv pure\ (\lambda f \rightarrow f\ x)\ \texttt{<*>}\ u$$

Leyes de functores aplicativos

Identidad:

$$pure\ id\ \texttt{<*>}\ u \equiv u$$

Composición:

$$pure\ (.)\ \texttt{<*>}\ u\ \texttt{<*>}\ v\ \texttt{<*>}\ w \equiv u\ \texttt{<*>}\ (v\ \texttt{<*>}\ w)$$

Homomorfismo:

$$pure\ f\ \texttt{<*>}\ pure\ x \equiv pure\ (f\ x)$$

Intercambio:

$$u\ \texttt{<*>}\ pure\ x \equiv pure\ (\lambda f \rightarrow f\ x)\ \texttt{<*>}\ u$$

Si se cumplen, entonces se cumple:

$$fmap\ f\ x = pure\ f\ \texttt{<*>}\ x$$

Funciones sobre functores aplicativos

$$
\begin{aligned}
&\mathit{sequenceA} :: \mathit{Applicative}\ f \Rightarrow [f\ a] \to f\ [a] \\
&\mathit{sequenceA}\ [\,] \quad\ = \mathit{pure}\ [\,] \\
&\mathit{sequenceA}\ (a : as) = (:) \texttt{<\$>}\ a \texttt{<*>}\ \mathit{sequenceA}\ as
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
&\mathit{traverse}\ f = \mathit{sequenceA}.\mathit{fmap}\ f
\end{aligned}
$$

que equivale a:

$$
\begin{aligned}
&\mathit{traverse}\ f\ [\,] \quad\ = \mathit{pure}\ [\,] \\
&\mathit{traverse}\ f\ (x : xs) = (:) \texttt{<\$>}\ f\ x \texttt{<*>}\ \mathit{traverse}\ f\ xs
\end{aligned}
$$

Alternative

En *Control.Applicative* también se define:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
  some  :: f a → f [a]   -- one or more
  many  :: f a → f [a]   -- zero or more
```

Ejemplo de Alternative: Parsers

```
instance Applicative (Parser s) where
  pure = pSucceed
  <*>  = <*>

instance Alternative (Parser s) where
  empty  = pFail
  <|>    = <|>
  many   = pList
  some p = (:) <$> p <*> pList p
```

donde

```
  pFail   :: Parser s a
pSuceed   :: a → Parser s a
  <*>     :: Parser s (a → b) → Parser s a → Parser s b
  <|>     :: Parser s a → Parser s a → Parser s a
  pList   :: Parser s a → Parser s [a]
```

Ejemplo: listas

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f ← fs, x ← xs]
```

Ejemplo:

$$[(+1),(+2)] \texttt{<*>} [1,2,3]$$

Ejemplo: listas

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f ← fs, x ← xs]
```

Ejemplo:

$$[(+1),(+2)] \texttt{<*>} [1,2,3]$$

leadsto

$$[2,3,4,3,4,5]$$

Ejemplo: Either

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

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left  e
```

Ejemplo: Either

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

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left  e
```

Una posible instancia de *Applicative*:

```
instance Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Right f <*> Left e  = Left e
  Left e  <*> _       = Left e
```

Ejemplo: Either

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

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left  e
```

Una posible instancia de *Applicative*:

```
instance Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Right f <*> Left e  = Left e
  Left e  <*> _       = Left e
```

Otra:

```
instance Monoid e ⇒ Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Left e  <*> Right _ = Left e
  Right _ <*> Left e  = Left e
  Left e  <*> Left e' = Left (e `mappend` e')
```

Composición

La clase de functores aplicativos es cerrada bajo la composición.

```
newtype (f :. g) a = Compose {getCompose :: f (g a)}

instance (Functor f, Functor g) ⇒ Functor (f :. g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)

instance (Applicative f, Applicative g)
                ⇒ Applicative (f :. g) where
  pure x = Compose (pure (pure x))
  Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

Composición

La clase de functores aplicativos es cerrada bajo la composición.

```
newtype (f :. g) a = Compose {getCompose :: f (g a)}

instance (Functor f, Functor g) ⇒ Functor (f :. g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)

instance (Applicative f, Applicative g)
                ⇒ Applicative (f :. g) where
  pure x = Compose (pure (pure x))
  Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

La composición de dos mónadas puede no ser una mónada, pero es un aplicativo.