Análisis de Lenguajes de Programación

Trabajo Practico Final: *list-func-solver*

Bautista Marelli, *M-6682/6*

16 de septiembre de 2021

Índice

1. Motivación **2**
1.1. ¿Que son las funciones sobre listas? 2

2. Instalación **3**

3. Uso **3**

4. TAD de FList **5**

5. Trabajo a Futuro **7**

1. Motivación

La evaluación y prueba de una función de lista puede resultar demasiado tediosa de hacer en en papel y lápiz. *list-func-solver* es un lenguaje de dominio especifico que nos permite evaluar las funciones de listas. Las características que más resaltan en este lenguaje son la posibilidad de guardar variables y funciones definidas en un enviroment y también la posibilidad de evaluar las funciones de listas utilizando distintas instancias (como por ejemplos utilizando Secuencias o la estructura de datos que el usuario desee implementar).

1.1. ¿Que son las funciones sobre listas?

En primer lugar, definimos que entendemos por una lista. Decimos que una lista es una secuencia ordenada y finita de cero o más elementos de $\mathbb{N}_0$. Llamamos $\mathcal{L}$ al conjunto de las listas y notamos al conjunto de listas con k elementos como $\mathcal{L}^k$.
Las funciones de listas son las funciones que van de $\mathcal{L}$ en $\mathcal{L}$. Tenemos definidas algunas funciones base y sus dominios:

- Cero a Izquierda: $\mathcal{L}$
$$O_i[1,2,3] = [\mathbf{0},1,2,3]$$
- Cero a Derecha: $\mathcal{L}$
$$O_r[1,2,3] = [1,2,3,\mathbf{0}]$$
- Borrar a Izquierda: $\mathcal{L}^{\geq 1}$
$$\square_i[\mathbf{1},2,3] = [2,3]$$
- Borrar a Derecha: $\mathcal{L}^{\geq 1}$
$$\square_r[1,2,\mathbf{3}] = [1,2]$$
- Sucesor a Izquierda: $\mathcal{L}^{\geq 1}$
$$S_i[\mathbf{1},2,3] = [\mathbf{2},2,3]$$
- Sucesor a Derecha: $\mathcal{L}^{\geq 1}$
$$S_r[1,2,\mathbf{3}] = [1,2,\mathbf{4}]$$

También podemos definir la composición de funciones de listas. Dadas dos funciones de listas F, G y una lista X, construimos la composición $H = FG$ como

$$HX = G[FX] \tag{1}$$

Por último, sea $F : \mathcal{L} \to \mathcal{L}$, definimos la **repetición** de F, que lo denotamos $\{F\}$ como:

$$\{F\}[x,Y,z] = \begin{cases} [x,Y,z] & x = z \\ F\{F\}[x,Y,z] & x \neq z \end{cases} \tag{2}$$

En palabras, la función $\{F\}$ actúa aplicando F hasta que el primer y el último elementos de su argumento son iguales. Notemos que $dom(\{F\}) = \mathcal{L}^{\geq 2}$.

2. Instalación

Un requisito previo para utilizar el programa es tener instalado `stack`. Luego los pasos a seguir son:

- Descargar el código del proyecto:

  ```
  git clone https://github.com/BMarelli/list-func-solver.git
  ```

- Acceder a la carpeta del proyecto:

  ```
  cd list-func-solver
  ```

- Inicializar *stack*:

  ```
  stack setup
  ```

- Construir el entorno:

  ```
  stack build
  ```

3. Uso

Para utilizar el programa se debe ejecutar de la siguiente manera:

```
stack exec list-func-solver-exe
```

```
bautistamarelli@BM list-func-solver (master) stack exec list-func-solver-exe
Evaluador de Funciones de Listas.
:? o :help para conocer los comandos y como utilizar el programa.
Se abrio el archivo Ejemplos/Prelude.fl correctamente!.
FL> _
```

Para conocer los comandos, se utiliza `:?` o `:help`. Dentro del programa, podemos realizar las siguientes operaciones:

- Evaluar expresiones. En el caso que queramos utilizar una instancia particular de la lista, utilizamos `<type>` para instanciar la expresión con la instancia `type`

  ```
  exp <type>;
  ```

- Definir funciones:

  ```
  def name = ... ;
  ```

- Definir variables:

  ```
  const name = exp <type>;
  ```

- Inferir la longitud de una expresión:

  ```
  exp :: nat ;
  ```

- Cargar archivos con funciones y variables definidas. El archivo debe terminar con `.fl`:

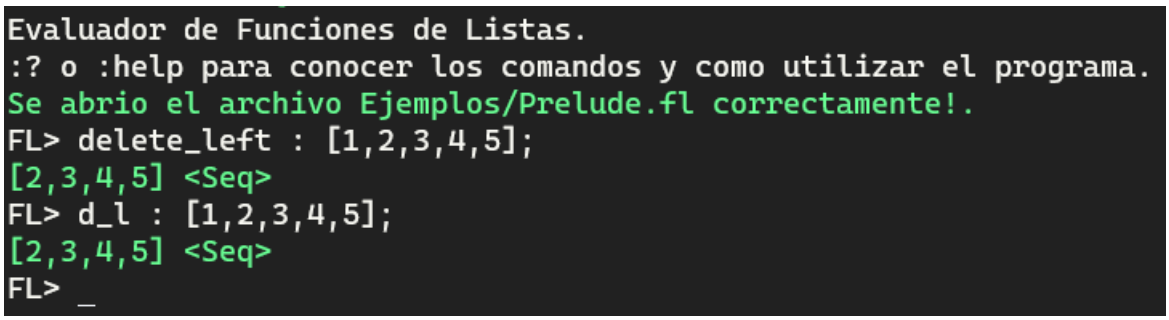
```
:load file.fl
```

Como podemos ver, es importante que las operaciones terminen con un `;`. Vale la pena mencionar que las expresiones están compuestas por 2 partes. La primera es la secuencia de funciones y la segunda es la lista o variable guardada en un enviroment. Estas partes están separadas por un `:`. Por ejemplo, si queremos aplicar la funcion `delete_left` a la lista `[1,2,3,4,5]`, la expresión nos queda:

```
delete_left : [1,2,3,4,5];
```

Las funciones bases también pueden ser abreviadas. En el caso de `delete_left` la podemos abreviar como `d_l` y en el ejemplo anterior nos queda:

```
d_l : [1,2,3,4,5];
```



4. TAD de FList

A continuación daremos la especificación del comportamiento de nuestro TAD de FList (el TAD de las lista de funciones de listas) y también explicaremos como agregar una nueva instancia al programa.
Podemos dar la especificación de las siguientes operaciones:

```
tad FList (A :: Set) where
    import Nat, String, Either, Error
    lengthFL :: FList A -> Nat
    fromList :: List A -> FList A
    quote :: FList A -> List A
    printFL :: FList A -> String

    zero :: Orientation -> FList A -> FList A
    succesor :: Orientation -> FList A -> Either Error (FList A)
    delete :: Orientation -> FList A -> Either Error (FList A)
    rep :: List Funcs -> FList A -> Either Error (FList A)

lengthFL <> = 0
lenghtFL <x1, ..., xn> = n
fromList [] = <>
fromList [x1, ..., xn] = <x1, ..., xn>
quote <> = []
quote <x1, ..., xn> = [x1, ..., xn]
printFL <> = "<>"
printFL <x1, ..., xn> = "<x1, ..., xn>"

zero L <> = <0>
zero R <> = <0>
zero L <x1, ..., xn> = <0, x1, ..., xn>
zero R <x1, ..., xn> = <x1, ..., xn, 0>
succesor L <> = Left Error
succesor R <> = Left Error
succesor L <x1, ..., xn> = Right <1 + x1, ..., xn>
succesor R <x1, ..., xn> = Right <x1, ..., 1 + xn>
delete L <> = Left Error
delete R <> = Left Error
delete L <x1, ..., xn> = Right <x2, ..., xn>
delete R <x1, ..., xn> = Right <x1, ..., xn-1>
rep (f:fs) <> = Left Error
rep (f:fs) <x1> = Left Error
rep (f:fs) <x1, ..., xn> = Right <x1, ..., xn> si x1 == xn, n >= 2
rep (f:fs) <x1, ..., xn> = rep (f:fns) ((f:fs) <x1, ..., xn>)
                           si x1 != xn, n >= 2
```

Para agregar una nueva especificación, tenemos que seguir los siguientes pasos:

1. Crear un nuevo archivo dentro de la carpeta `src/List` donde vamos a definir la nueva especificación

2. Importar los archivos `src/AST.hs` y `src/List/FList.hs`

```
module List.NuevaInstancia where
import AST
import List.FList
```

3. Definir la instancia utilizando la estructura que queríamos instanciar. Por ejemplo con la estructura X (`data X a = ...`)

```
data X a = ...
instance FList X where
...
```

4. En el archivo `src/AST.hs` crear una nueva referencia en la estructura `Type`. También agregar la nueva instancia al mapa de los tipos de estructuras

```
data Type = ... | TNew

mapType = M.fromList [..., ("NuevaInstancia", TNew)]
```

5. En el archivo `src/ListEval.hs` importamos el archivo donde definimos la nueva instancia. También en la función `aplicar` agregamos una nueva línea para la nueva instancia `TNew`

```
import List.NuevaInstancia
...
aplicar fs xs TNew = do l <- aplicar' @X fs xs
                        return (quote l, TNew)
```

5. Trabajo a Futuro

En esta sección hablaremos de las cosas que se pueden agregar al proyecto de forma que este sea mas amplio.
Uno de los complementos muy importante para este proyecto es la notación potencia. En funciones de listas, la notación potencia nos permite expresar operaciones de forma mas sencilla y legible. Primero definamos esta notación. Dada una función de lista F, denotamos a la potencia k-esima como

$$F^k[x_1, ..., x_n] = \begin{cases} [x_1, ..., x_n] & k = 0 \\ F^{k-1}F[x_1, ..., x_n] & k > 0 \end{cases} \tag{3}$$

Esta definición de la notación potencia nos permite representar de una manera sencilla a muchas de las funciones recursivas numéricas (FR).

- Funciones cero $c^{(n)}$: $F_c =$ `zero_left`
- Funciones proyección $p_k^{(n)}$: $F_{p_k} =$ `(move_right)^k-1 double_left (swap move_left)^k-1`
- Funciones sucesor s: $F_s =$ `double_left succ_left`
- Operador composición: Sean las funciones numéricas $f^{(n)}$ y $\{g_i^{(k)}\}_{i=1}^n$ con representaciones en FRL por Ff y $\{Fgi\}_{i=1}^n$, la composición de las funciones $h = \Phi(f, g_1, ..., g_n)$ la podemos expresar de la siguiente manera:

  ```
  H = Fg1 move_right Fg2 move_right ... Fgn (move_left)^n-1 Ff
      (move_right delete_left)^n move_left
  ```

- Operador recursión: Sean las funciones numéricas $g^{(n)}$ y $h^{(k+2)}$ con representaciones en FRL por Fg y Fh. Entonces la función definida por recursión $f = R(g, h)$ la podemos expresar de la siguiente manera:

  ```
  F = move_right Fg zero_left
    { move_right swap move_left Fh move_right delete_left succ_left swap move_left }
      delete_left swap move_left
  ```

- Operador minimización: Sea la función numérica $h^{(n+1)}$ con representaciones en FRL por Fh. Entonces la función obtenida por minimización de h, $g(X) = M[h](X) = \mu_t(h(t, X) = 0)$ la podemos expresar de la siguiente manera:

  ```
  F = zero_left Fh zero_right { delete_left succ_left Fh } delete_left delete_right
  ```

De esta forma, con estas definiciones, podemos expresar todas las funciones recursivas numéricas de una manera simple. Esto nos permite utilizar este proyecto no solo para evaluar funciones de listas, sino también evaluar las representaciones de las funciones que pertenecen al conjunto de FR.