Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación – 2° cuat. 2008

Fecha de entrega: 18 de Septiembre

Intérprete de lenguajes imperativos

El objetivo de este trabajo práctico consiste en interpretar lenguajes usando una semántica operacional basada en reglas. Vamos a definir una sentencia como una transformación de estados, y un combinador como una función que dadas dos sentencias devuelve otra, combinándolas de alguna forma.

```
type Sentence s = s -> s
type Combinator s = Sentence s -> Sentence s -> Sentence s
```

Para la resolución del TP, considerar que pueden resultar útiles algunas de las siguientes funciones definidas en el preludio: `foldr`, `map`, `iterate`, `filter`, `concatMap`, `dropWhile`, `head`, `tail`.

Operadores genéricos

Un programa será una lista de sentencias que se ejecutan en algún orden (según lo indique la regla de la secuencia).

1. Escribir el combinador `(>->)`, que, dadas dos sentencias, las combina en una nueva sentencia que ejecuta primero la de la izquierda, y luego la de la derecha.

   `(>->) :: Combinator s`

   Por ejemplo, con `s = String`:

   `(shl >-> dup) "abc"` $\rightsquigarrow$ `"bbccaa"` [1]

2. Escribir el combinador `(<-<)`, que combina las sentencias para que se ejecuten en orden inverso al de `(>->)`, es decir primero la de la derecha y luego la de la izquierda.

   `(<-<) :: Combinator s`

   Por ejemplo:

   `(shl <-< dup)  "abc"` $\rightsquigarrow$ `"abbcca"`

[1] `shl` y `dup` se explican más adelante

3. Escribir la regla de la secuencia, que, dada una lista de sentencias y un combinador, combina todas las sentencias de la lista, para ejecutarlas de acuerdo a lo que indique el combinador. Asumir que el combinador en cuestión es asociativo.

   `secAbs :: Combinator s -> [Sentence s] -> Sentence s`

   Por ejemplo:

   `(secAbs (<-<) [shl,shl,dup]) "abc"` $\rightsquigarrow$ `"bbccaa"`

4. Escribir la regla de la secuencia estándar, que usa el combinador (`>->`). Dada una lista de sentencias, esta operación permite combinarlas para ejecutarlas en orden.

   `sec :: [Sentence s] -> Sentence s`

5. Escribir el generador de combinadores, que, dado un combinador `c` y una sentencia `s`, devuelve un nuevo combinador que se comporta de forma similar a `c`, pero que ejecuta `s` antes de cada sentencia.

   `(><) :: Combinator s -> Sentence s -> Combinator s`

   Por ejemplo:

   `((>->) >< shl) shl dup "abcd"` $\rightsquigarrow$ `"ddaabbcc"`

6. Escribir el generador de combinadores de bifurcación. Si `pred` es un predicado, entonces `(bifurcacion pred)` es un combinador que toma dos sentencias, y elige una u otra dependiendo del resultado de evaluar el predicado en el estado inicial.

   `bifurcacion :: (s -> Bool) -> Combinator s`

   Por ejemplo, si `s = String`, puede tomarse un predicado de modo tal que:

   `pred :: String -> Bool`

   `bifurcacion pred :: Combinator String`

   Tomando `pred = (\x -> head x == 'b')` se tiene que:

   `bifurcacion pred shl dup "abc"` $\rightsquigarrow$ `"aabbcc"`

Lenguaje PP

El lenguaje PP utiliza un estado de `String`, y cuenta con las sentencias shl, shr, dup y forN.

```
type PPState = String
type PPSentence = Sentence PPState
```

7. Escribir la regla de `shl`, que es una sentencia que resulta en un *shift left* en el estado del programa.

   `shl :: PPSentence`

   Por ejemplo:

   `shl "abc"` $\rightsquigarrow$ `"bca"`

8. Escribir la regla de `shr`, análoga a `shl` pero hacia la derecha.

   `shr :: PPSentence`

   Por ejemplo:

   `shr "abc"` $\rightsquigarrow$ `"cab"`

9. Escribir la regla de `dup`, que duplica todos los elementos en el estado del programa.

   `dup :: PPSentence`

   Por ejemplo:

   `dup "abc"` $\rightsquigarrow$ `"aabbcc"`

10. Escribir la regla de `forN`, que dado un natural n y un bloque de código, devuelve la sentencia asociada a ejecutar n veces el bloque de código, según la regla de secuencia estándar (i.e. las sentencias se ejecutan de izquierda a derecha, en orden).

    `forN :: Int -> [PPSentence] -> PPSentence`

    Por ejemplo:

    `forN 2 [shl, dup] "abc"` $\rightsquigarrow$ `"bbccccaaaabb"`

Lenguaje Jay

El lenguaje Jay utiliza como estado una lista (finita) de duplas `(String,Int)`, que representa una asignación de un valor entero para cada una de las variables de un programa.[2] Una expresión es una función que dado un estado devuelve un valor. Por ejemplo, la expresión $x + 2$ tiene sentido siempre y cuando se disponga de un estado que provea un valor para x. Las sentencias, del mismo modo que antes, transforman un estado en otro.

Las sentencias y las expresiones del lenguaje Jay se definen de la siguiente manera:

```
type JayState = [(String, Int)]
type JayExpression t = JayState -> t
type JaySentence = Sentence JayState
```

Se definen las siguientes funciones para crear expresiones básicas (constantes y variables) en Jay:

```
con :: Int -> JayExpression Int
con c s = c

var :: String -> JayExpression Int
var varName s = maybe 0 id (lookup varName s)
```

[2]Para que la lista de duplas represente una función total, se harán las dos consideraciones siguientes: (1) si una variable no figura en la lista de duplas, el valor asociado será 0; (2) la lista no tiene duplas que refieran a la misma variable.

De este modo, la expresión `con 10` representa la constante 10. El valor de esta expresión no depende del estado en el que se la evalúe. Por ejemplo:

`con 10 [("x", 42), ("y", 28)]` $\rightsquigarrow$ `10`

Por su parte, la expresión `var "y"` representa la variable "y". Cuando se evalúa esta expresión en un estado, se devuelve el valor asociado a "y". Por ejemplo:

`var "y" [("x", 42), ("y", 28)]` $\rightsquigarrow$ `28`

11. Escribir la función `asig` que representa la regla de asignación.

    ```
    asig :: String -> JayExpression Int -> JaySentence
    ```

    Por ejemplo:

    ```
    asig "foo" (con 28) [("bar",42),("quux",1)]
    ⇝ [("foo",28),("bar",42),("quux",1)]
    ```

    ```
    asig "bar" (var "foo") [("bar",42),("foo",28),("quux",1)]
    ⇝ [("bar",28),("foo",28),("quux",1)]
    ```

12. Escribir la función `op` que representa la regla de evaluación de operadores.

    ```
    op :: (a -> b -> c) -> JayExpression a -> JayExpression b -> JayExpression c
    ```

    Por ejemplo, tomando `a = Int`, `b = Int` y `c = Bool` con el operador `(==)`, se tiene:

    ```
    (==) :: Int -> Int -> Bool
    ```

    ```
    op (==) :: JayExpression Int -> JayExpression Int -> JayExpression Bool
    ```

    Con lo cual se obtendría un comportamiento como el siguiente:

    ```
    op (==) (var "foo") (var "bar") [("foo",28),("bar",42),("quux",1)]
    ⇝ False
    ```

    ```
    op (==) (var "foo") (con 28) [("foo",28),("bar",42),("quux",1)]
    ⇝ True
    ```

13. Escribir la función `rIf` que representa la regla del if. Utilizar el combinador de bifurcación.

    ```
    rIf :: JayExpression Bool -> [JaySentence] -> [JaySentence] -> JaySentence
    ```

    Por ejemplo:

    ```
    rIf (op (>) (var "x") (var "y"))
         [asig "max" (var "x")]     -- rama then
         [asig "max" (var "y")]     -- rama else
       [("x",15),("y",12)]          -- estado inicial


    ⇝ [("max",15),("x",15),("y",12)]
    ```

14. Escribir la función `rWhile`, que representa la regla del while. Utilizar el combinador de bifurcación. Sugerencia: utilizar la función `iterate` definida en el preludio.

    ```
    rWhile :: JayExpression Bool -> [JaySentence] -> JaySentence
    ```

    En el siguiente ejemplo, se computa `y` = 2^4. Para ello se comienza con el estado `[("x",4), ("y",1)]`. En cada paso se decrementa `x` y se duplica `y`:

```
rWhile (op (>) (var "x") (con 0))
     [asig "x" (op (-) (var "x") (con 1)),
      asig "y" (op (*) (var "y") (con 2))]
     [("x",4),("y",1)]   -- estado inicial
```

$\rightsquigarrow$ `[("x",0),("y",16)]`

Ejemplos de ejecución

Asignación simple

El siguiente es un ejemplo de la ejecución de "`x := x + 1`" partiendo del estado `[("x",41)]`:

- `asig "x" (op (+) (var "x") (con 1))` es de tipo `JaySentence`, es decir, es una función que transforma un estado en otro.
- Por lo tanto, la sentencia aplicada a un estado inicial debe devolver un nuevo estado. Por ejemplo, se quiere que:

  `asig "x" (op (+) (var "x") (con 1)) [("x",41)]`
  $\rightsquigarrow$ `[("x",42)]`
- Por empezar, se evalúa la expresión `(op (+) (var "x") (con 1))`. Recordemos que una expresión es una función que, dado un estado, devuelve un valor. La expresión se aplica entonces al estado inicial:

  `op (+) (var "x") (con 1) [("x",41)]`
  $\rightsquigarrow$ `(var "x" [("x",41)]) + (con 1 [("x",41)])` $\rightsquigarrow$ `41 + (con 1 [("x",41)])`
  $\rightsquigarrow$ `41 + 1`
  $\rightsquigarrow$ `42`
- Una vez obtenido el valor, `asig` reemplaza en la lista el viejo valor de `x` por el valor nuevo. El resultado es final es el estado `[("x",42)]`, de acuerdo a lo deseado.

Ejecución en secuencia

A continuación se muestra un ejemplo de la ejecución de dos asignaciones en secuencia:

- La siguiente sentencia:

  `asig "x" (con 42) >-> asig "y" (var "x")`

  surge de combinar "`x := 42`" y "`y := x`" mediante el combinador `(>->)`. El resultado final debe ser un estado en el cual los valores de `x` e `y` son ambos `42`, sin importar cuál fuera el valor inicial de `x`.
- Apliquemos por ejemplo la sentencia completa al estado `[("x",10)]`:

  `(asig "x" (con 42) >-> asig "y" (var "x")) [("x",10)]`

  El resultado esperado sería:

  `[("y",42),("x",42)]`

- El combinador `(>->)` evaluará primero la sentencia de la izquierda con el estado inicial:

  ```
  asig "x" (con 42) [("x",10)]
  ⇝ [("x",42)]
  ```

- Una vez obtenido este nuevo estado, el combinador `(>->)` evaluará la sentencia de la derecha *en el estado nuevo*:

  ```
  asig "y" (var "x") [("x",42)]
  ⇝ [("y",42),("x",42)]
  ```

  Y este es efectivamente el resultado esperado.

Pautas de entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas y **enviarlo por correo electrónico a la lista de docentes**. Cada función debe contar con un comentario donde se explique su funcionamiento. Junto con el código impreso se debe entregar el código. El mismo debe poder ser ejecutado en `Hugs`. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté adecuadamente comentado.

Los objetivos a evaluar en la implementación de las funciones son:

- Corrección.
- Declaratividad.
- Reuso de funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso de funciones de alto orden, currificación y listas por comprensión.
- **No utilizar recursión explícita**. La idea del TP es aprender a aprovechar las características enumeradas en el ítem anterior.