

Práctica 6 Interpretación

NOTA. En todos los ejercicios suponemos que el lenguaje de implementación del intérprete cuenta con:

- Un tipo **Id** para representar identificadores (nombres de variables).
- Un tipo **Env a** de los entornos que asocian identificadores a valores de tipo **a**, con las siguientes operaciones:
 - $\emptyset :: \text{Env } a$ denota un entorno vacío.
 - Si $E :: \text{Env } a$ es un entorno y $x :: \text{Id}$ es un identificador entonces $E(x) :: a$ es el valor asociado a x en el entorno E .
 - Si $E :: \text{Env } a$ es un entorno, $x :: \text{Id}$ es un identificador y $v :: a$ es un valor de tipo **a**, entonces $E[x := v] :: \text{Env } a$ es el entorno extendido asociando x a v .
- Un tipo **Addr** para representar direcciones de memoria.
- Un tipo **Memory a** de las memorias que en cada dirección de memoria almacenan un valor de tipo **a**, con las siguientes operaciones:
 - $\emptyset :: \text{Memory } a$ denota una memoria vacía.
 - Si $M :: \text{Memory } a$ es una memoria y $a :: \text{Addr}$ es una dirección, entonces $M(a) :: a$ es el valor almacenado en la dirección a en la memoria M .
 - Si $M :: \text{Memory } a$ es una memoria, $\text{allocate}(M) :: \text{Addr}$ es una dirección de memoria disponible.
 - Si $M :: \text{Memory } a$ es una memoria, $a :: \text{Addr}$ es una dirección, y $v :: a$ es un valor de tipo **a**, entonces $M[a := v] :: \text{Memory } a$ es la memoria extendida almacenando v en a .
- Una operación *error* que siempre falla.

Las operaciones de acceso a memorias y entornos son parciales (pueden fallar). Por último, utilizamos notación estilo Haskell porque es concisa y precisa, pero supondremos siempre que el lenguaje de implementación es call-by-value (los argumentos de las funciones se evalúan por completo antes de la invocación a la función).

Ejercicio 1. Dado un tipo **FuncName** que representa nombres de funciones, consideremos el siguiente lenguaje. Una definición de función está dada por un identificador de función, una lista de parámetros formales, y el cuerpo de la función que es una expresión. Las expresiones son las que se enumeran abajo:

```

data Definition = Define FuncName [Id] Expr
data Expr = ExprVar Id -- x (variable)
          | ExprConstNum Int -- n (número)
          | ExprAssign Id Expr -- x := e (asignación)
          | ExprSeq Expr Expr -- e1; e2 (secuencia)
          | ExprAdd Expr Expr -- e1 + e2 (suma)
          | ExprCall FuncName [Expr] -- f(e1, ..., eN) (llamado a función)
          | ExprIfZero Expr Expr Expr -- if==0 e1 then e2 else e3

```

La construcción `ExprIfZero` evalúa `e1`. Si su valor es 0 evalúa `e2`; de lo contrario evalúa `e3`. Los valores en el lenguaje son números enteros. La evaluación de todos los argumentos se hace de izquierda a derecha.

1. Definir un intérprete que recibe una expresión, la lista de definiciones de funciones, el entorno inicial, la memoria inicial, y devuelve el valor que resulta de evaluar la expresión, junto con el estado final de la memoria:

```

eval :: Expr -> [Definition] -> Env Addr -> Memory Int
      -> (Int, Memory Int)

```

2. Evaluar el programa $f(3)$ con la siguiente definición:

```

define f(n) =
  if==0 n
  then 0
  else
    ((n := n + -1); f(n)) + n

```

Observar que `n + -1` es la suma de la variable `n` con la constante `-1`.

Resultado esperado: 3.

Ejercicio 2. Consideremos el tipo `Cont` de las continuaciones que a partir de un valor construyen un resultado final:

```

type Cont a = a -> Result

```

En este contexto `Val` es el tipo de los valores que se obtienen cuando se evalúa una expresión del lenguaje, y `Result` es el tipo de los “resultados finales”. No nos interesa explicitar cuáles son los resultados finales: vamos a tratarlos como un tipo abstracto.

La siguiente es la sintaxis abstracta de un lenguaje con números enteros, booleanos y tuplas:

```

data Expr =
  ExprConstNum Int -- n (número)
| ExprAdd Expr Expr -- e1 + e2 (suma)
| ExprConstBool Bool -- b (booleano)
| ExprIf Expr Expr Expr -- if e1 then e2 else e3 (if)
| ExprTuple [Expr] -- (e1, ..., eN) (tupla)
| ExprProj Int Expr --  $\pi_n(e)$  (proyección)

```

Los valores posibles son los siguientes:

```

data Val = VNum Int
          | VBool Bool
          | VTuple [Val]

```

1. Definir un intérprete que reciba una expresión y una continuación, y devuelva un resultado final:

```
eval :: Expr -> Cont Val -> Result
```

2. Evaluar la siguiente expresión, suponiendo que `Result` y `Val` son el mismo tipo, y que $k_0 :: \text{Cont Val}$ es la continuación definida como $k_0(x) = x$:

```
if True
then  $\pi_2(1 + 1, 2 + 2, 3 + 3)$ 
else 9
```

Resultado esperado: 4.

Ejercicio 3. Como en el ejercicio anterior, dado el tipo de las continuaciones:

```
type Cont a = a -> Result
```

consideremos el lenguaje dado por la siguiente sintaxis abstracta:

```
data Expr = ExprConstNum Int      -- n          (número)
          | ExprAdd Expr Expr      -- e1 + e2      (suma)
          | ExprFail               -- fail         (fail)
          | ExprAmb Expr Expr      -- e1 | e2      (amb)
```

En este lenguaje la evaluación de una expresión puede terminar con éxito o fallar.

- La expresión `(ExprConstNum n)` devuelve n y siempre termina con éxito.
- La expresión `(ExprAdd e_1 e_2)` devuelve la suma de los números denotados por e_1 y e_2 y termina con éxito si ambas subexpresiones terminan con éxito.
- La expresión `ExprFail` siempre falla.
- La expresión `(ExprAmb e_1 e_2)` denota la “alternativa no determinística”, cuyo significado es el siguiente: evaluar e_1 y en caso de éxito devolver su resultado. En caso de que la evaluación de e_1 falle, evaluar e_2 y devolver su resultado.

1. Definir un intérprete que recibe una expresión, dos continuaciones, y devuelve un resultado final:

```
eval :: Expr -> Cont Int -> Cont () -> Result
```

La primera continuación recibe un número y es la que se debe invocar en caso de éxito. La segunda continuación recibe siempre el parámetro `()` y es la que se debe invocar en caso de falla.

2. Evaluar la expresión:

```
(fail | 2) + (3 | fail)
```

Resultado esperado: 5

Ejercicio 4. Para el cálculo- λ :

```
data Expr =
  ExprVar Id           -- x
| ExprLambda Id Expr   --  $\lambda x.e$ 
| ExprApp Expr Expr     -- e1 e2
```

Definir los siguientes intérpretes:

1. Usando la estrategia call-by-value (evaluando los argumentos antes que el cuerpo de las funciones):

```
evalByValue :: Expr -> Env Val -> Val
```

Los entornos asocian identificadores a valores. Los valores en este caso son clausuras.

```
data Val = VClosure Id Expr (Env Val)
```

2. Usando la estrategia call-by-name (copiar los argumentos sin evaluar):

```
evalByName :: Expr -> Env Thunk -> Val
```

Los entornos asocian identificadores a *thunks*: un *thunk* es una expresión no evaluada acompañada de un entorno que define todas sus variables libres:

```
data Thunk = Thunk Expr (Env Thunk)
```

Los valores son clausuras:

```
data Val = VClosure Id Expr (Env Thunk)
```

3. Usando la estrategia call-by-need (o *lazy*, en los que se mantienen copias a los argumentos). Este intérprete utiliza la técnica de *store-passing*, es decir, recibe una memoria como tercer parámetro y devuelve una memoria como parte de su resultado:

```
evalByNeed :: Expr -> Env Addr -> Memory Val -> (Val, Memory Val)
```

Los valores en este caso son clausuras o *thunks*:

```
data Val = VClosure Id Expr (Env Val)
        | VThunk Expr (Env Val)
```

Las variables pueden estar ligadas a *thunks*, pero se espera que el resultado de evaluar una expresión nunca sea un *thunk*.

Ejercicio 5. Considerar el lenguaje dado por la siguiente sintaxis abstracta:

```

data Expr =
  ExprVar Id          -- x          (variable)
| ExprLambda Id Expr  -- λx.e       (función anónima)
| ExprApp Expr Expr   -- e1 e2      (aplicación)
| ExprConstNum Int    -- n          (constante numérica)
| ExprAdd Expr Expr   -- e1 + e2    (suma)
| ExprConstBool Bool  -- b          (constante lógica)
| ExprIf Expr Expr Expr -- if e1 then e2 else e3
                        --           (condicional)
| ExprAssign Id Expr  -- x := e     (asignación)
| ExprSeq Expr Expr   -- e1; e2     (secuencia)

```

1. Definir un intérprete que use la estrategia de evaluación call-by-value y la técnica de *store-passing*:

```
eval :: Expr -> Env Addr -> Memory Val -> (Val, Memory Val)
```

Los valores en este caso son numéricos, booleanos o clausuras:

```

data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Val)

```

2. Evaluar la siguiente expresión:

```

let suma = λx.λy.((x := x + y); x) in
let f = suma 10 in
let x = 1 in
  f 2

```

Resultado esperado: 12.

Ejercicio 6. Dado un tipo `Message` que representa nombres de mensajes, consideramos la siguiente sintaxis abstracta para un lenguaje orientado a objetos:

```

data Expr =
  ExprNew          -- new
| ExprExtend Expr Message Id Expr -- extend e1.m(x) = { e2 }
| ExprSend Expr Message Expr    -- e1.m(e2)
| ExprLet Id Expr Expr          -- let x = e1 in e2
| ExprSeq Expr Expr            -- e1; e2

```

- La expresión `new` denota un nuevo objeto.
- La expresión `extend e1.m(x) = { e2 }` extiende dinámicamente el objeto denotado por la expresión `e1`. Define un método de tal manera que su respuesta al mensaje `m` con el parámetro `x` sea el resultado de evaluar la expresión `e2`. Por convención, esta expresión denota el valor de `e1`.
- La expresión `e1.m(e2)` envía el mensaje `m` al objeto denotado por `e1` dándole como argumento el objeto denotado por `e2`. El resultado final de evaluar esta expresión es la respuesta que devuelve `e1`.

- La expresión `let x = e1 in e2` representa una declaración local (no recursiva).
- La expresión `e1; e2` es la composición en secuencia de las expresiones `e1` y `e2`. El valor de `e1` se descarta y se devuelve el de `e2`.

El intérprete recibe un entorno (`Env Addr`) que asocia variables a direcciones de memoria y una memoria (`Memory Val`) que asocia direcciones de memoria a valores.

Los valores posibles en este lenguaje son únicamente objetos. Cada objeto posee un diccionario que asocia mensajes a clausuras léxicas que incluyen: el nombre del parámetro, el cuerpo del mensaje, y el entorno en el que fueron definidos.

```
data Val = VObject (Dictionary Message (Id, Expr, Env Addr))
```

1. Definir un intérprete usando la estrategia de evaluación call-by-value y la técnica de *store-passing*:

```
eval :: Expr -> Env Addr -> Memory Val -> (Val, Memory Val)
```

2. Verificar que al evaluar la siguiente expresión en el entorno vacío y con memoria vacía el intérprete entra en un ciclo infinito:

```
let obj = new in
  extend obj.ciclo(x) = { obj.ciclo(x) };
  obj.ciclo(obj)
```