Un lenguaje funcional puro sencillo (SFL) y su intérprete

Eduardo Bonelli

Departamento de Computación
FCEyN
UBA

3 de octubre, 2006

Intérprete para un lenguaje funcional sencillo (SFL)

- Componentes de un intérprete
- Sintaxis concreta y abstracta de SFL
- Intérprete
- Extensiones
  1. binding local (`let`)
  2. procedimientos (`proc`)
  3. recursión (`letrec`)

Componentes de un intérprete

- Definición inductiva de
  - las expresiones sintácticamente correctas (*sintaxis concreta*)
  - la representación interna de las expresiones (*sintaxis abstracta*)
- Designar los valores expresados: subconjunto de expresiones a los que puede evaluar una expresión arbitraria
- Un entorno ("environment"): estructura de datos que liga variables con valores denotados
- Un entorno *inicial*
- Intérprete como función recursiva sobre el conjunto inductivo de las expresiones (ie. de la sintaxis abstracta)
- Salida del intérprete: valores expresados

Sintaxis Concreta de SFL

La sintaxis de SFL viene dada por la siguiente gramática BNF:

```
<program> ::= <expr>

<expr> ::= <number> | <identifier>
         | <primitive> ({<expr>*(,)})
         | if <expr> then <expr> else <expr>

<primitive> ::= + | - | * | add1 | sub1 | isZero
```

Ejemplos

Ejemplos de expresiones válidas:

```
3
n
+(3,n)
add1(add1(3,n),4)
if -(2,+(3,4)) then 2 else 3
+(if 2 then 3 else 3,7)
```

NOTA: `if` testea valores expresados

- **falso**: 0
- **verdadero**: cualquier valor expresado distinto de 0

Sintaxis abstracta

```
data Program = Pgm Exp

data Exp = LitExp Int |
           VarExp Symbol |
           IfExp Exp Exp Exp |
           PrimApp PrimOp [Exp]

data PrimOp = AddPrim | SubtractPrim | MultPrim | IncrPrim |
              DecrPrim | ZeroTestPrim

type Symbol = String
```

Mapeo de Sintaxis Concreta a Abstracta

- El mapeo de sintaxis concreta a sintaxis abstracta se realiza en dos pasos:
  1. Análisis léxico ("lexer")
  2. Análisis sintáctico ("parser")
- La cátedra los proveerá de modo que puedan probar el intérprete
- Para el ejemplo: `let x=3 in *(2,x)`
- salida del lexer: `[TokenLet,TokenVar "x",TokenEq,TokenInt 3,TokenIn,TokenTimes, TokenOB, TokenInt 2, TokenComma, TokenVar "x", TokenCB]`
- salida del parser: `Pgm (LetExp ["x"] [LitExp 3] (PrimApp MultPrim [LitExp 2,VarExp "x"]))`

Mapeo de Sintaxis Concreta a Abstracta

- El mapeo de sintaxis concreta a sintaxis abstracta se realiza en dos pasos:
  1. Análisis léxico ("lexer")
  2. Análisis sintáctico ("parser")
- La cátedra los proveerá de modo que puedan probar el intérprete
- Para el ejemplo: `let x=3 in *(2,x)`
- salida del lexer: `[TokenLet,TokenVar "x",TokenEq,TokenInt 3,TokenIn,TokenTimes, TokenOB, TokenInt 2, TokenComma, TokenVar "x", TokenCB]`
- salida del parser: `Pgm (LetExp ["x"] [LitExp 3] (PrimApp MultPrim [LitExp 2,VarExp "x"]))`

Mapeo de Sintaxis Concreta a Abstracta

- El mapeo de sintaxis concreta a sintaxis abstracta se realiza en dos pasos:
  1. Análisis léxico ("lexer")
  2. Análisis sintáctico ("parser")
- La cátedra los proveerá de modo que puedan probar el intérprete
- Para el ejemplo: `let x=3 in *(2,x)`
- salida del lexer: `[TokenLet,TokenVar "x",TokenEq,TokenInt 3,TokenIn,TokenTimes, TokenOB, TokenInt 2, TokenComma, TokenVar "x", TokenCB]`
- salida del parser: `Pgm (LetExp ["x"] [LitExp 3] (PrimApp MultPrim [LitExp 2,VarExp "x"]))`

Valores

Valores expresados

Valores que puede arrojar la evaluación de una expresión.

- Ej. números, procedimientos

Valores denotados

Valores que se ligan a los identificadores (ie. a las variables).

- Ej. números, procedimientos
- Más adelante: referencias

NOTA: Con la simple descripción del conjunto de valores expresados y denotas ya se dice bastante sobre el lenguaje objeto.

Intérpete de programas

El valor expresado resultante de evaluar `+(3,4)` es claro.

- ¿Qué sucede con la expresión `+(x,5)`?
- ¿A qué valor expresado debe evaluar?

Es necesario contar con un entorno: estructura de datos que liga variables con valores denotados
Dado el entorno que liga el valor 2 a la variable, podemos responder que `+(x,5)` debe evaluar al valor 7.
Asumiremos que el intérprete comienza su ejecución contando con un entorno inicial.

Entornos

- Representación de entornos en Haskell

```
data Env = EmptyEnv |
           ExtendedEnvRecord [Symbol]
                             [ExpressedValue]
                             Env

data ExpressedValue = Nro {stripNro::Int}

-- extender entorno con nuevo rib
extendEnv::[Symbol]->[ExpressedValue]->Env->Env
extendEnv ids vals env = ExtendedEnvRecord ids vals env
```

Entornos

- Buscar el valor expresado asociado a una variable

```
applyEnv::Env->Symbol->ExpressedValue
applyEnv EmptyEnv sym =
        error ("applyEnv: Sin binding para "++ show sym)
applyEnv (ExtendedEnvRecord syms vals env) sym
     = case findInList syms sym of
       (Just n) -> vals!!n
       _ -> applyEnv env sym

-- buscar simbolo en lista de simbolos (retornar posicion)
findInList::[a]->a->Maybe Int
```

Intérpete de programas

```
-- interprete de programas
evalProgram (Pgm body) = evalExpr body initEnv

-- entorno inicial
initEnv = ExtendedEnvRecord ["i","v","x"]
                            [Nro 1, Nro 5, Nro 10]
                            EmptyEnv
```

Interpretando expresiones

```
evalExpr (LitExp n) env = Nro n
evalExpr (VarExp id) env = applyEnv env id
evalExpr (PrimApp prim rands) env =
      let args = map (\x->evalExpr x env) rands
        in applyPrimitive prim args
evalExpr (IfExp testExp trueExp falseExp) env =
     if isTrueValue (evalExpr testExp env)
        then evalExpr trueExp env
        else evalExpr falseExp env

isTrueValue::ExpressedValue->Bool
isTrueValue = not . (==0) . stripNro
```

Aplicando primitivas

```
applyPrimitive AddPrim args =
    Nro (stripNro (args!!0) + stripNro (args!!1))
applyPrimitive SubtractPrim args =
    Nro (stripNro (args!!0) - stripNro (args!!1))
applyPrimitive MultPrim args =
    Nro (stripNro (args!!0) * stripNro (args!!1))
applyPrimitive IncrPrim args = Nro (stripNro (args!!0) + 1)
applyPrimitive DecrPrim args = Nro (stripNro (args!!0) - 1)
applyPrimitive ZeroTestPrim  args =
    Nro (if stripNro (args!!0)==0 then 1 else 0)
```

Declaraciones locales a través de ejemplos

```
let x = 5,
    y = 6
in +(x,y)

-- dependencia entre declaraciones

let x = 3
  in let y = +(x,4)
         in *(x,y)

let x = 3,
    y = +(x,4)
in *(x,y)
```

Declaraciones locales a través de ejemplos

```
-- declaraciones internas enmascaran las externas

let x = 3
in let x = *(x,x)
   in +(x,x)
```

Declaraciones locales

```
let var1 = exp1,
    ...
    varn-1 = expn-1,
    varn = expn
  in cuerpo
```

Sintaxis concreta

<expr> ::= let {<ident> = <expr>}$^{*(,)}$ in <expr>

Región La región asociada con var1,...,varn es cuerpo

Evaluación

- cada expi (en ningún orden en particular!)
- ligar cada vari al valor de expi
- evaluar cuerpo y retornar su resultado como el resultado de toda la expresión.

Procedimientos a través de ejemplos

```
let f = proc (y,z) +(y,-(z,5))
  in (f 2 28)

(proc (y,z) +(y,-(z,5)) 2 28)

-- si bien no tenemos letrec, podemos hacer esto
let
  factAux = proc(next,x)
              if zero?(x) then 1
                          else (next next sub1(x))
in
  let fact = proc(x) (factAux factAux x)
  in (fact 3)
```

Procedimientos: ejemplos

Sintaxis concreta

```
<expr> ::= ... | proc ({<identifier>}*(,)) <expr>
           | (<expr> {<expr>}*)
```

Sintaxis abstracta

```
data Exp = LitExp Int |
           VarExp Symbol |
           IfExp Exp Exp Exp |
           PrimApp PrimOp [Exp] |
           LetExp [Symbol] [Exp] Exp |
           ProcExp [Symbol] Exp |
           AppExp Exp [Exp]
```

Extendiendo el intérprete (`let`)

```
evalExpr (LitExp n) env = Nro n
...
evalExpr (LetExp ids rands body) env =
        let args = map (\x-> evalExpr x env) rands
         in
          evalExpr body (extendEnv ids args env)
```

Extendiendo el intérprete (proced.)

¿A qué valor debería evaluar un procedimiento?
Ejemplo:

```
let x = 5
 in let f = proc (y,z) +(y, -(z,x))
        x = 28
     in (f 2 x)
```

Observar: el cuerpo es evaluado

- solamente cuando es aplicado
- en un entorno que liga los parámetros formales a los argumentos de la aplicación

¿Y qué hay de las variables que ocurren libres en el procedimiento?

Evaluación de procedimientos

Ejemplo:

```
let x = 5
 in let f = proc (y,z) +(y, -(z,x))
        x = 28
     in (f 2 x)
```

Respecto a las variables que ocurren libres en el procedimiento

- deben respetar scoping léxico (o estático)
  - deben evaluarse usando el entorno en efecto en el momento de la creación
- Por lo tanto, debemos retener (junto con el cuerpo y los parámetros formales) las valores asociados a las variables libres en el momento en el que el procedimiento fue creado.

Clausuras

- cuerpo junto con un entorno y parámetros formales

```
data ExpressedValue = Nro {stripNro::Int} |
                      Closure [Symbol] Exp Env
```

- Volviendo a los valores expresados/denotados:
  - **Valores expresados = Números + Clausuras**
  - **Valores denotados = Números + Clausuras**

Extendiendo el intérprete

```
evalExpr (ProcExp ids body) env = Closure ids body env
evalExpr (AppExp rator rands) env =
        let proc =  evalExpr rator env
            args =  map (\x->evalExpr x env) rands
         in
          case proc of
            (Closure _ _ _) -> applyProcVal proc args
            _ -> error ("evalExpr: Intento de aplicar un
                         no-procedimiento "++show proc)

applyProcVal (Closure ids body env) args =
        evalExpr body (extendEnv ids args env)
```

Extendiendo el intérprete para SFL con recursión

- Ejemplos
- Sintaxis concreta y abstracta
- Implementando recursión a través de la creación de clausuras durante el lookup (V1)
  - Entornos en notación de sintaxis abstracta (anexando un nuevo tipo de rib)
  - Entornos en notación procedural
- Implementando recursión armando clausura una única vez (V2)
  - Entornos en notación de sintaxis abstracta (utilizando estructuras circulares)

Ejemplos en SFL con recursión

- ```
  letrec
    fact(x) = if zero?(x)
                then 1
                else *(x,(odd sub1(x)))
  in (fact 6)
  ```
- ```
  letrec
    even(x) = if zero?(x)
                then 1
                else (odd sub1(x))
    odd(x)  = if zero?(x)
                then 0
                else (even sub1(x))
  in (odd 13)
  ```

Sintaxis concreta y abstracta

Sintaxis concreta

```
<expr> ::=
    letrec
      {<identifier> ({<identifier>}*(,)) = <expr>}*
      in <expr>
```

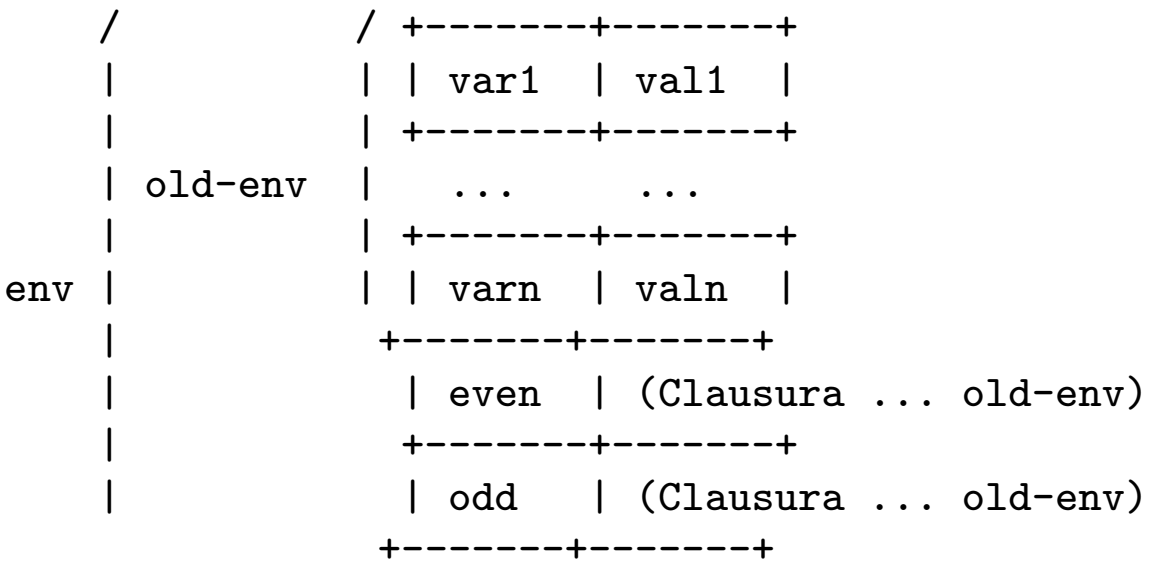
Sintaxis abstracta

```
data Exp = LitExp Int |
           ... |
           LetRecExp [Symbol] [[Symbol]] [Exp] Exp
```

Implementación: versión ingenua

```
letrec
 even(x) = if zero?(x) then 1 else (odd sub1(x))
 odd(x)  = if zero?(x) then 0 else (even sub1(x))
in (odd 13)
```

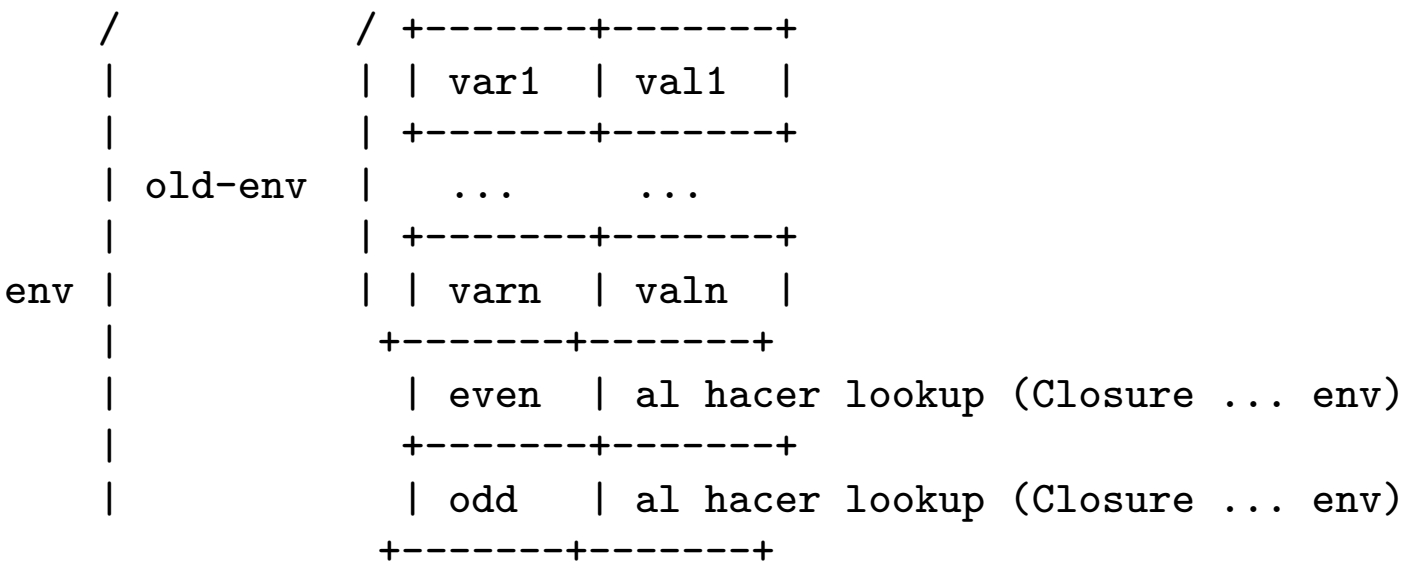
Manipulación ingenua de entornos y clausuras:



Clausuras deben referenciar **env** (y no **old-env**). Circularidad!

Implementación: Creación de clausuras al hacer lookup

- Idea: demorar creación de clausuras hasta que el procedimiento recursivo es llamado



Creación de clausuras al hacer Lookup - v1

- En ésta versión los entornos se representan como un tipo algebraico
- Se agrega un nuevo constructor para extender entornos con declaraciones recursivas locales

```
data Env = EmptyEnv |
           ExtendedEnvRecord [Symbol]
                             [ExpressedValue]
                             Env |
           RecursivelyExtendedEnvRecord [Symbol]
                                        [[Symbol]]
                                        [Exp]
                                        Env
```

Creación de clausuras al hacer Lookup - v1

- El código para `evalExpr` es el esperado

```
evalExpr (LetRecExp procNames idss bodies letrecBody) env =
       (evalExpr letrecBody
          (extendEnvRecursively procNames idss bodies env))

extendEnvRecursively procNames idss bodies env =
          RecursivelyExtendedEnvRecord procNames idss bodies env
```

- Lo interesante es cómo se realiza el lookup de un procedimiento recursivo en el entorno...

Creación de clausuras al hacer Lookup - v1

```
applyEnv::Env->Symbol->ExpressedValue
applyEnv EmptyEnv sym =
      error ("applyEnv: Sin binding para "++ show sym)
applyEnv (ExtendedEnvRecord syms vals env) sym
     = case findInList syms sym of
       (Just n) -> vals!!n
       _ -> applyEnv env sym
applyEnv env@(RecursivelyExtendedEnvRecord procNames idss bodies
     = case findInList procNames sym of
       (Just n) -> Closure (idss!!n) (bodies!!n) env
       _ -> applyEnv oldEnv sym
```

Creación de clausuras al hacer Lookup - v2

- Entornos representados como funciones

```
empty-env sym =
     error ("empty-env: Sin binding para "++ show sym)

extend-env syms vals env sym =
      let pos = findInList sym syms
       in case pos of
            (Just n) -> vals!!n
            _ -> apply-env env sym

apply-env env sym = env sym
```

Creación de clausuras al hacer Lookup - v2

- Agregamos una nueva operación para extender entornos

```
extend-env-recursively proc-names idss bodies old-env =
  let rec-env sym =
     let pos = findInList sym proc-names
       in case pos of
           (Just n) -> Closure idss!!pos bodies!!pos rec-env
           _ -> apply-env old-env sym
   in rec-env
```

Creación de clausuras al hacer Lookup - v2

- `evalExpr`: sin cambios respecto a v1

```
evalExpr (LetRecExp procNames idss bodies letrecBody) env =
      (evalExpr letrecBody
         (extendEnvRecursively procNames idss bodies env))

extendEnvRecursively procNames idss bodies env =
    RecursivelyExtendedEnvRecord procNames idss bodies env
```

Creación única de clausuras

- En los dos casos previos las clausuras se creaban a la hora de hacer lookup
- Esto no es necesario dado que el entorno de la clausura creada siempre es el mismo
- Utilizando estructuras circulares podemos limitarnos a construir clausuras una sola vez
- Asumimos representación estándar de entornos

```
data Env = EmptyEnv |
           ExtendedEnvRecord [Symbol]
                             [ExpressedValue]
                             Env
```

Implementación: Creación única de clausuras

- El evaluador de expresiones es el mismo que antes

```
evalExpr (LetRecExp procNames idss bodies letrecBody) env =
       (evalExpr letrecBody
          (extendEnvRecursively procNames idss bodies env))
```

- Pero las clausuras se crean al procesar la declaración

```
extendEnvRecursively procNames idss bodies oldEnv =
    let
      env = ExtendedEnvRecord procNames lst oldEnv
      lst = map (\(xs,y) -> Closure xs y env) (zip idss bodies)
     in  env
```