Extendiendo SFL con asignación (SFLA) (SFLA = Prog. Funcional Imperativa)

Eduardo Bonelli

Departamento de Computación
FCEyN
UBA

3 de octubre, 2006

Expresiones con efectos

SFL es un lenguaje funcional **puro**

- Toda expresión se evalúa exclusivamente para obtener su valor

Sin embargo, a veces es necesario utilizar recursos imperativos (principalmente por cuestiones de eficiencia) como:

- Asignación de valores a variables
- Actualización in-place de estructuras de datos
- Operaciones de Entrada/Salida

Todas estas expresiones se evalúan para obtener un efecto

Referencias

- Hoy retomamos SFL, nuestro lenguaje funcional puro
- En SFL el valor asignado a un identificador (o variable) no puede alterarse; en este sentido las variables de SFL se asemejan a las variables usadas en Matemáticas o en Lógica
- En SFL a las variables se les asignaban valores expresados (valores que retornan las expresiones al ser evaluadas)
- Esto se resume como sigue

- Valores denotados=Number + Procval
- Valores expresados=Number + Procval

Referencias

- Hoy vamos a extender SFL con una operación de asignación
- La expresión `set x = e` representa la operación de asignación
  - Primero se evalúa `e` para obtener un valor `v`
  - Luego se obtiene la dirección de memoria asociada a `x`
  - Finalmente se asigna `v` al contenido de la dirección de `x`
- Observar que en `set x = +(x,2)`
  - La ocurrencia de `x` a la derecha del "=" se refiere a su contenido ("r-value" de `x`)
  - La ocurrencia de `x` a la izquierda del "=" se refiere a su dirección de memoria ("l-value" de `x`)

Referencias

- Ahora a las variables se les asignan

- referencias a una dirección mutable de memoria
- cuyo contenido puede ser modificado a través la operación de asignación

- Por lo tanto, ahora tendremos

- Valores denotados = **ReferenciaA**(Number + Procval)
- Valores expresados = Number + Procval

Ejemplos en SFL con asignación

Extendemos la sintaxis concreta de SFL con una operación de asignación (SFLA):

```
<expr> ::= ... | set <identifier> = <expr>
```

Ejemplo

```
let y=3
  in let dummy = set y = 4
     in y
```

- Evalúa a 4

Ejemplos en SFL con asignación

Pregunta: ¿A qué valor debe evaluar la siguiente expresión?

```
let x = 2
  in set x = 3
```

Respuesta:

- a ninguno (o bien a cualquier valor arbitrario que designemos)
- la expresión se evalúa exclusivamente por su efecto (el de cambiar el contenido de x)
- NO se espera ningún valor como *resultado* de la evaluación de una asignación
- Asumimos (arbitrariamente) que toda asignación evalúa a 0

Ejemplos en SFL con asignación

Pregunta: ¿A qué valor debe evaluar la siguiente expresión?

```
let x = 2
  in set x = 3
```

Respuesta:

- a ninguno (o bien a cualquier valor arbitrario que designemos)
- la expresión se evalúa exclusivamente por su efecto (el de cambiar el contenido de `x`)
- NO se espera ningún valor como *resultado* de la evaluación de una asignación
- Asumimos (arbitrariamente) que toda asignación evalúa a 0

Representación de estado local

Pregunta: ¿A qué valor debe evaluar la siguiente expresión?

```
let count =
  let localState = 0
  in proc () let dummy = set localState = add1(localState)
              in localState
in +((count),(count))
```

Respuesta:

- `count` mantiene una variable privada `localState`
- esta variable privada representa su estado local
- `localState` lleva la cuenta de cuántas veces fue llamado
- la expresión evalúa a 3

Representación de estado local

Pregunta: ¿A qué valor debe evaluar la siguiente expresión?

```
let count =
  let localState = 0
  in proc () let dummy = set localState = add1(localState)
              in localState
in +((count),(count))
```

Respuesta:

- `count` mantiene una variable privada `localState`
- esta variable privada representa su estado local
- `localState` lleva la cuenta de cuántas veces fue llamado
- la expresión evalúa a 3

Modalidades de pasaje de parámetros

Ejemplos

```
let x=100
  in let p = proc (y) let d = set y = add1(y) in y
     in +((p x),(p x))
```

- Hay dos modalidades de pasaje de parámetros bien diferenciadas
  - Pasaje por valor: crear una nueva referencia (ie. una copia) por cada parámetro formal
  - Pasaje por referencia: pasar la referencia existente como parámetro en lugar de una copia

Modalidades de pasaje de parámetros

Ejemplos

```
let x=100
  in let p = proc (y) let d = set y = add1(y) in y
     in +((p x),(p x))
```

- Hay dos modalidades de pasaje de parámetros bien diferenciadas
  - Pasaje por valor: retorna 202
  - Pasaje por referencia: retorna 203

Secuenciamiento

- En la presencia de expresiones que arrojan efectos el orden de evaluación es crucial
- Considere la siguiente expresión

```
let y=2
 in +(y,let d = set y = 4 in y)
```

- Si los argumentos se evalúan de izquierda a derecha, arroja 6 como resultado
- Si los argumentos se evalúan de derecha a izquierda, arroja 8 como resultado

Secuenciamiento

- Como consecuencia, un mecanismo explícito de secuenciamiento es necesario
- Extendemos la sintaxis concreta de SFL

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

- En la expresión `(begin e1 e2 ... en)`
  - las expresiones `e1`, `e2`, ..., `en` se evalúan en *ese orden*
  - los valores retornados por `e1`, `e2`, ..., `en-1` son descartados
  - el valor retornado por `en` es el resultado de *toda la expresión*

Secuenciamiento

- El siguiente ejemplo evalúa a `4`

```
let y = 2
in (begin set y = 4, y)
```

- El siguiente ejemplo evalúa a `0`. El valor de la primera expresión argumento de `begin` (ie. `y`) se descarta.

```
let y = 2
in (begin y, set y = 4)
```

Sintaxis abstracta

```
data Program = Pgm Exp

data Exp = LitExp Int | VarExp Symbol | ...
         | SetExp Symbol Exp | BeginExp [Exp]

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

type Symbol = String
```

Entornos y stores

El intérprete ahora toma como entrada

1. una expresión a evaluar
2. un entorno que mapea variables a referencias
3. una memoria (llamada "store") que mapea referencias (direcciones de memoria) a valores expresados (su contenido)

```
type Store = Reference -> ExpressedValue
```

Entornos

- El entorno mapea variables a referencias
  - Las referencias representan direcciones de memoria
  - Se implementan como enteros

```
type Reference = Int

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

Entornos - operaciones

- Creación de entorno vacío

```
emptyEnv :: Env
emptyEnv = EmptyEnv
```

- Extensión de un entorno con lista de variables

```
extendEnv :: [Symbol] -> Reference -> Env -> Env
extendEnv ids ref env =
    ExtendedEnvRecord ids [ref..ref+(length ids)-1] env
```

Entornos - operaciones

- Lookup en un entorno

```
applyEnv:: Env -> Store -> Symbol -> ExpressedValue
applyEnv env st sym = st (applyEnvRef env sym)

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

Store

- La memoria se llama store
- Un store mapea referencias a valores expresados

```
type Store = Reference -> ExpressedValue
```

- Asumimos que si `s` es un store, entonces `s` aplicado a `0` indica la siguiente referencia libre para ser asignada
- Vamos a ver algunas operaciones sobre stores

Store - operaciones

- Creación de un store vacío

```
emptyStore :: Store
emptyStore = \ref-> if ref==0
                       then Nro 1
                       else error "Store vacio!"
```

- Siguiente referencia disponible para ser asignada

```
nextAvail :: Store -> Reference
nextAvail s = stripNro (s 0)
```

Store - operaciones

- Actualizar contenido de una referencia

```
updateStore :: Store -> Reference ->
                  ExpressedValue -> Store
updateStore s ref a = let newStore r
                                | ref==r = a
                                | otherwise = s r
                      in newStore
```

Store - operaciones

- Asignar espacio nuevo en la memoria e inicializar

```
extendStoreUnit :: ExpressedValue -> Store -> Store
extendStoreUnit a s = let nextPos = nextAvail s
                      in let newStore ref
                                    | ref==0 = Nro (nextPos+1)
                                    | ref==nextPos = a
                                    | otherwise = s ref
                         in newStore

extendStore :: Store -> [ExpressedValue] -> Store
extendStore s xs = foldr extendStoreUnit s (reverse xs)
```

Intérprete

- Antes el resultado de evaluar una expresión consistía sólo en un valor expresado (número o clausura)

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

- Ahora una expresión puede tener un efecto (alterar la memoria)
- El resultado de evaluar una expresión consiste en
  - Un valor expresado (número o clausura)
  - Un store: el estado resultante de la memoria

```
data Answer = AnAnswer {value::ExpressedValue, store::Store}
```

Intéprete - Main

- Intérprete de programas

```
evalProgram :: Program -> Answer
evalProgram (Pgm body) =
        evalExpression body emptyEnv emptyStore
```

- Intérprete de expresiones

```
evalExpression :: Exp -> Env -> Store -> Answer
evalExpression (LitExp n) env st = AnAnswer (Nro n) st

evalExpression (VarExp id) env st =
         AnAnswer (applyEnv env st id) st
```

Intérprete - PrimApp

- Operaciones primitivas

```
evalExpression (PrimApp prim rands) env st =
    let (args,s1) =
         foldl (\(xs,s) x-> let ans=evalExpression x env s
                            in ((value ans):xs,store ans))
               ([],st)
               rands
    in AnAnswer (applyPrimitive prim (reverse args)) s1
```

Intérprete - IfExp

- Interpretando if-then-else

```
evalExpression (IfExp testExp trueExp falseExp) env st =
     let ans = evalExpression testExp env st
       in if isTrueValue (value ans)
            then evalExpression trueExp env (store ans)
            else evalExpression falseExp env (store ans)
```

Intérprete - Asignación

- Interpretando la asignación

```
evalExpression (SetExp id rhs) env st =
   let ans=evalExpression rhs env st
     in let ref=applyEnvRef env id
         in AnAnswer (Nro 0)
                     (updateStore (store ans) ref (value ans))
```

  - El valor expresado resultante de evaluar una asignación es `(Nro 0)`
  - Es el valor que fijamos (arbitrariamente) al principio de la clase
  - De interés es el store actualizado que es retornado

Intérprete - Secuenciamiento

- Interpretando begin

```
evalExpression (BeginExp exps) env st =
    let (args,s1) =
         foldl (\(xs,s) x-> let ans=evalExpression x env s
                            in ((value ans):xs,store ans))
               ([],st)
               exps
    in if (null args)
          then AnAnswer (Nro 0) s1
          else AnAnswer (head args) s1
```