Semántica de los Conceptos Fundamentales de los Lenguajes de Programación

Profesor: Juan Francisco Diaz
jdiaz@eisc.univalle.edu.co

Asistente de Docencia: Gerardo M. Sarria M.
gsarria@eisc.univalle.edu.co

May 17, 2002

Ligadura Local

La forma `let` es una expresión que sirve para crear una variable ligada. Su sintaxis concreta es:

⟨expresión⟩ ::= `let` {⟨identificador⟩ = ⟨expresión⟩}* `in` ⟨expresión⟩
`let-exp (ids rands body)`

Con la que se pueden crear expresiones como:

```
let x = 1
in let x = +(x,2)
   in add1(x)
```

En este ejemplo la referencia de `x` en la primera aplicación se refiere a la declaración de más afuera, mientras que la referencia a la `x` en la segunda aplicación se refiere a la declaración de más adentro, y por lo tanto el valor de toda la expresión es 4.

Ligadura Local (cont.)

La sintaxis abstracta de una expresión es ahora:

```
(define-datatype expression expression?
  (lit-exp
    (datum number?))
  (var-exp
    (id symbol?))
  (primapp-exp
    (prim primitive?)
    (rands (list-of expression?)))
  (if-exp (test-exp true-exp false-exp)
    (if (true-value?  (eval-expression test-exp env))
      (eval-expression true-exp env)
      (eval-expression false-exp env)))
  (let-exp
    (ids (list-of symbol?))
    (rands (list-of expression?))
    (body expression?)))
```

Ligadura Local (cont.)

Una expresión `let` es evaluada de la siguiente forma:

1. Se evalúa la subexpresión correspondiente a las declaraciones, en el ambiente original.
2. Se extiende el ambiente con las variables de la declaración y sus valores.
3. Se evalúa el cuerpo de la expresión `let` en el ambiente extendido.

Ligadura Local (cont.)

El comportamiento de las ligaduras locales en el interpretador se obtiene agregando la cláusula `let-exp` a `eval-expression`:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args)))
      (if-exp (test-exp true-exp false-exp)
        (if (true-value?  (eval-expression test-exp env))
          (eval-expression true-exp env)
          (eval-expression false-exp env)))
      (let-exp (ids rands body)
        (let ((args (eval-rands rands env)))
          (eval-expression body (extend-env ids args env)))))))
```

Procedimientos

Para que el interpretador tenga más poder de expresión, se necesitan crear procedimientos. La siguiente sintaxis se usa para crear procedimientos y aplicaciones:

⟨expresión⟩ ::= `proc` ({⟨identificador⟩}$^{*(,)}$) ⟨expresión⟩
`proc-exp (ids body)`

::= (⟨expresión⟩ {⟨expresión⟩}*)
`app-exp (rator rands)`

De esta manera se pueden crear programas como

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

Los valores expresados y denotados, cambian con la inclusión de procedimientos:

Valor Expresado = Valor Denotado = Número + ProcVal

donde ProcVal es el conjunto de valores que representan los procedimientos.

Procedimientos (cont.)

Cuando un procedimiento es aplicado, su cuerpo es evaluado en el ambiente en que fué creado, extendido con la ligadura de los parámetros formales del procedimiento a los argumentos de la aplicación.

Por tanto, las variables que ocurren libres en el procedimiento se evalúan en el ambiente que envuelve al procedimiento.

Ejemplo 1:

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

Cuando se llama a `f`, su cuerpo debe ser evaluado en un ambiente que liga `y` a 2, `z` a 28 y `x` a 5.
`x` es ligado a 5 ya que el alcance de la declaración interna no incluye la declaración del procedimiento.

Procedimientos (cont.)

Para que un procedimiento retenga la ligadura que sus variables libres tienen en el momento que es creado, éste debe ser *empaquetado*, para que pueda ser usado independiente del ambiente donde es llamado. Éste paquete es llamado *clausura* (*closure*), y corresponderá al nuevo conjunto de valores ProcVal.

ProcVal se puede pensar como un tipo de dato. La interfaz se puede definir como un árbol de sintaxis abstracta representado por

```
(define-datatype procval procval?
  (closure
    (ids (list-of symbol?))
    (body expression?)
    (env environment?)))
```

En esta representación, `apply-procval` usa `cases` para tomar la clausura e invocar el cuerpo de la clausura en el ambiente extendido:

```
(define apply-procval
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body (extend-env ids args env))))))
```

Procedimientos (cont.)

Ahora, se modificará `eval-expression` para que maneje procedimientos. El código del cliente sólo puede manipular los procedimientos mediante la interfaz de ProcVal, de modo que sea independiente de la representación de los procedimientos.

Cuando un procedimiento es evaluado, se construye una clausura y se retorna inmediatamente.

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (proc-exp (ids body) (closure ids body env))
      ...)))
```

El cuerpo del procedimiento no es evaluado aquí: no puede ser evaluado hasta que los valores de los parámetros formales sean conocidos.

Procedimientos (cont.)

Cuando una aplicación es evaluada, el operador y los operandos son evaluados y el resultado es enviado a `apply-procval`.

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval?  proc)
            (apply-procval proc args)
            (eopl:error 'eval-expression
              "Attempt to apply non-procedure ~s" proc))))
      ...)))
```

Los operandos son denominados *parámetros actuales*. Ellos son expresiones y no deben ser confundidos con sus valores (llamados *argumentos* del procedimiento) o con las *variables ligadas* o *parámetros formales* del procedimiento.

Procedimientos (cont.)

Ejercicio:
Calcule el resultado de la siguiente expresión (se escribe $<< exp >>$ para denotar el árbol de sintaxis abstracta asociado con la expresión exp, y se escribe $[x = a, y = b]env$ en lugar de `(extend-env '(x y) '(`a b`) `env`)`).

```
(eval-expression <<let x = 5
                   in let x = 38
                          f = proc (y, z) *(y, +(x, z))
                          g = proc (u) +(u, x)
                      in (f (g 3) 17)>>
                 env0)
```

Recursión

Para agregar la recursión en el interpretador, se tomará la forma de las expresiones `let` cambiando su espacio de declaración y limitándolo a expresiones tipo `proc` de esta manera::

⟨expresión⟩ ::= `letrec`
{⟨identificador⟩ ({⟨identificador⟩}$^{*(,)}$) = ⟨expresión⟩}*
`in` ⟨expresión⟩
`letrec-exp (proc-names idss bodies letrec-body)`

Ejemplo 2:

```
letrec
  fact(x) = if zero?(x) then 1 else +(x, (fact sub1(x)))
in (fact 6)
```

Recursión (cont.)

Para evaluar una expresión `letrec`, se evalúa el cuerpo de la expresión en un ambiente extendido con las declaraciones dentro de la misma expresión recursiva:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (letrec-exp (proc-names idss bodies letrec-body)
        (eval-expression letrec-body
          (extend-env-recursively proc-names idss bodies env)))
      ...)))
```

Recursión (cont.)

El Tipo Abstracto de Dato `environment` es modificado para admitir una nueva variante:

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vals (list-of scheme-value?)
    (env environment?))
  (recursively-extended-env-record
    (proc-names (list-of symbol?))
    (idss (list-of (list-of symbol?)))
    (bodies (list-of expression?))
    (env environment?)))
```

Recursión (cont.)

El comportamiento de `(apply-env` e' `name)` para la nueva variante del tipo `environment` es el siguiente:
Sea e' `(extend-env-recursively proc-names idss bodies` e`)`, entonces

1. Si `name` es uno de los nombres en `proc-names`, e `ids` y `body` son la lista de parámetros formales y el cuerpo del procedimiento (respectivamente), entonces `(apply-env` e' `name) = (closure ids body` e'`)`.
2. Si no, entonces `(apply-env` e' `name) = (apply-env` e `name)`.

```
(define extend-env-recursively
  (lambda (proc-names idss bodies old-env)
    (recursively-extended-env-record
      proc-names idss bodies old-env)))
```

Recursión (cont.)

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'empty-env "No binding for ~s" sym))
      (extended-env-record (syms vals old-env)
        (let ((pos (list-find-position sym syms)))
          (if (number?  pos)
            (list-ref vals pos)
            (apply-env old-env sym))))
      (recursively-extended-env-record (proc-names idss
                                        bodies old-env)
        (let ((pos (list-find-position sym proc-names)))
          (if (number?  pos)
            (closure
              (list-ref idss pos)
              (list-ref bodies pos)
              env)
            (apply-env old-env sym)))))))
```

Asignación de Variables

Una expresión `set` asigna un valor a una variable. La forma `begin` permite la evaluación de expresiones en orden retornando el último valor. La sintaxis concreta de las asignaciones y la secuenciación es:

⟨expresión⟩ ::= `set` ⟨identificador⟩ = ⟨expresión⟩
`varassign-exp (ids rhs-exp)`

::= `begin` ⟨expresión⟩ {`;` ⟨expresión⟩}* `end`
`begin-exp (exp exps)`

La asignación agrega la siguiente variante al tipo de dato expresión:

```
(varassign-exp
  (id symbol?)
  (rhs-exp expression?))
```

Asignación de Variables (cont.)

Al entrar la asignación, surge la pregunta: ¿Cuál es la diferencia entre asignación y ligadura?

Una ligadura, crea una nueva asociación de un nombre con un valor, mientras que la asignación cambia el valor de una ligadura existente.
Ligadura comprende la asociación de nombres con valores; asignación comprende el compartimiento de valores entre diferentes procedimientos.

Ejemplo 3:
Considere el siguiente programa:

```
let x = 0
in letrec
     even() = if zero?(x)
              then 1
              else begin set x = sub1(x); (odd) end
     odd()  = if zero?(x)
              then 0
              else begin set x = sub1(x); (even) end
   in begin set x = 13; (odd) end
```

Los procedimientos `even` y `odd` comparten la variable `x` y se comunican cambiando el estado de la variable, en vez de pasarse datos explicitamente.

Asignación de Variables (cont.)

Un ambiente es simplemente un tipo de dato que asigna valores a variables.
Cuando se incluyeron los procedimientos, se encontró el problema de que el estado del ambiente no era guardado en el momento de la aplicación de procedimientos y esto ocasionaba el retorno de un valor errado.

Ejemplo 4:

```
let z = 0
in let f = proc (y) *(y,z)
   in let z = 1
      in (f 2)
```

Este ejemplo en un principio evalúa a 2. Sin embargo, con la entrada de la clausura, el ambiente, en el momento de la aplicación del procedimiento, es guardado y la expresión retorna el valor correcto: 0.

Asignación de Variables (cont.)

Ahora, con la entrada de la asignación, surge otro problema.

Ejemplo 5:

```
let z = 0
    f = proc (x) z
in begin set z = 1;
         (f 2)
   end
```

La expresión anterior tiene el valor 0, en vez del valor 1.
Lo anterior debido a que en el momento de la asignación de `z` a 1, el valor de dicha variable en el ambiente no cambió; lo que cambió fué el valor de la asociación creada en la ligadura local. Luego en el llamado al procedimiento se guarda el ambiente (donde `z` tiene el valor de 0) y se retorna 0.

Asignación de Variables (cont.)

Para evitar el problema anterior, cada identificador debe denotar la dirección de una ubicación en memoria (la memoria es también llamada *store*). Dicha dirección se denomina referencia y lo que hace la asignación es modificar su contenido.

Los valores denotados son ahora referencias cuyos valores son valores expresados:

$$\begin{array}{lcl} \text{Valor Denotado} & = & \text{Ref(Valor Expresado)} \\ \text{Valor Expresado} & = & \text{Número} + \text{ProcVal} \end{array}$$

Las referencias o ubicaciones son tambien llamadas *L-valores* (valores que se asocian con variables que aparecen al lado izquierdo (*Left* en inglés) de la declaración de asignación).

De forma análoga, los valores expresados (que aparecen en el lado derecho (*Right*) de la declaración de asignación) son llamados *R-valores*.

Asignación de Variables (cont.)

En el lenguaje que se está realizando, se creará un nueva referencia para cada parámetro formal en cada llamado a procedimiento. Esta política es conocida como *llamado por valor*. Cuando se asigna a un parámetro formal, la asignación es local al procedimiento.

Ejemplo 4:

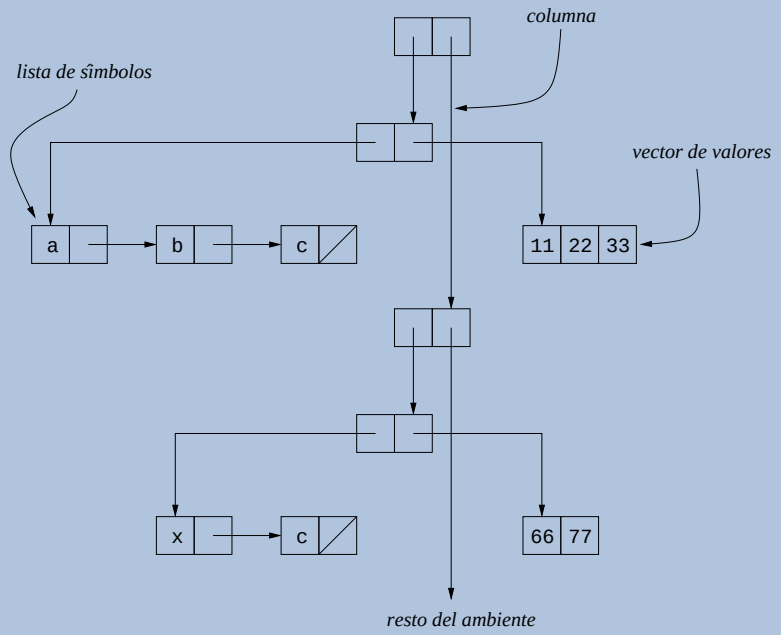
```
let x = 100
in let p = proc (x) let d = set x = add1(x)
                    in x
   in +((p x), (p x))
```

El programa anterior retorna 202, ya que una nueva referencia es creada para cada `x` en cada uno de los llamados a procedimiento.
Además, en cada llamado a procedimiento la asignación afecta sólo las ligaduras internas.

Asignación de Variables (cont.)

Antes de implementar la asignación, se recuerda el concepto de ambiente.

Un ambiente puede verse como la estructura de la siguiente figura:



Asignación de Variables (cont.)

El tipo de dato ambiente es entonces:

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vals (list-of scheme-value?))
    (env environment?)))

(define scheme-value? (lambda (v) #t))
```

Los procedimientos son los siguientes:

```
(define empty-env
  (lambda ()
    '()))

(define extend-env
  (lambda (syms vals env)
    (cons (cons syms (list-vector vals)) env)))
```

Asignación de Variables (cont.)

Por último, el procedimiento para aplicar un ambiente:

```
(define apply-env
  (lambda (env sym)
    (if (null? env)
      (eopl:error 'apply-env "No binding for ~s" sym)
      (let ((syms (car (car env)))
            (vals (cdr (car env)))
            (env (cdr env)))
        (let ((pos (rib-find-position sym syms)))
          (if (number? pos)
            (vector-ref vals pos)
            (apply-env env sym)))))))
```

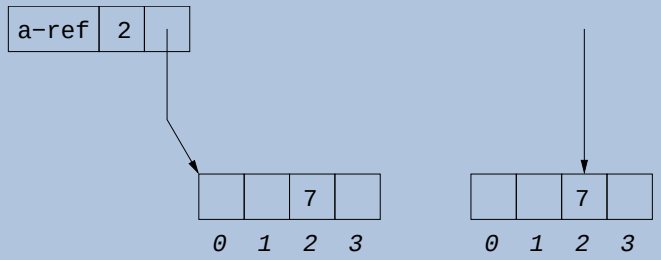
Asignación de Variables (cont.)

Para implementar la asignación de variables, se incluirá el tipo de dato `reference`, cuyas operaciones serán `deref` y `setref!`, que accesan y almacenan el valor de la ubicación.

El tipo de dato contiene un vector (las referencias serán elementos de un vector) y la posición del L-valor dentro de este vector:

```
(define-datatype reference reference?
  (a-ref
    (position integer?)
    (vec vector?)))
```

La siguiente figura muestra una referencia a una ubicación en un vector que contiene el 7. Se puede describir dicha referencia gráficamente dibujando un apuntador al centro de la estructura, como se muestra en el lado derecho del diagrama.



Asignación de Variables (cont.)

Las operaciones del tipo de dato se definen de la siguiente manera:

```
(define primitive-deref
  (lambda (ref)
    (cases reference ref
      (a-ref (pos vec) (vector-ref vec pos)))))

(define primitive-setref!
  (lambda (ref val)
    (cases reference ref
      (a-ref (pos vec) (vector-set! vec pos val)))))

(define deref
  (lambda (ref)
    (primitive-deref ref)))

(define setref!
  (lambda (ref val)
    (primitive-setref! ref val)))
```

Asignación de Variables (cont.)

Ahora, se debe revisar el concepto de ambiente para hacer uso de las referencias.
Se incluirá la operación `apply-env-ref` a la interfaz de ambiente para que cuando se encuentre un identificador, se retorne la referencia en vez de su valor.
El procedimiento `apply-env` se reescribirá en términos de `apply-env-ref` y `deref`:

```
(define apply-env
  (lambda (env sym)
    (deref (apply-env-ref env sym))))

(define apply-env-ref
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env-ref "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (rib-find-position sym syms)))
          (if (number? pos)
            (a-ref pos vals)
            (apply-env-ref env sym)))))))
```

Asignación de Variables (cont.)

Por último, para implementar la asignación de variables, simplemente se agrega la siguiente cláusula a `eval-expression`:

```
(varassign-exp (id rhs-exp)
  (begin
    (setref!
      (apply-env-ref env id)
      (eval-expression rhs-exp env))
    1))
```

Explícitamente se retorna 1, ya que lo que retorna `setref!` no está especificado, y siempre se tiene que retornar un valor expresado.

Paso de Parámetros

La forma más común para el paso de parámetros es el llamado por valor (visto anteriormente). Sin embargo, hay otras alternativas para el mecanismo de paso de parámetros.

Ejemplo 5:
Considere la siguiente expresion:

```
let a = 3
    p = proc (x) set x = 4
in begin (p a); a end
```

Bajo la semántica de llamado por valor, el valor denotado asociado con `x` es una referencia que inicialmente contiene el mismo valor que la referencia asociada con `a`. Sin embargo, la asignación a `x` no tiene efecto en el contenido de la referencia de `a`, por lo que el valor de toda la expresión es 3.

Paso de Parámetros (cont.)

Algunas veces es deseable que se permita pasar a un procedimiento variables con el objetivo de que éstas sean asignadas por dicho procedimiento (es decir, que el valor de las variables cambien tanto en el interior del procedimiento como en el llamado al procedimiento).

Lo anterior se realiza pasando al procedimiento una referencia a la ubicación de la variable y no su contenido. Este mecanismo es denominado *llamado por referencia*.

Usando el llamado por referencia en el ejemplo 5, la signación de `x` a 4 tendría el efecto de asignar 4 a `a`, por lo que toda la expresión retornaría 4 y no 3.

Paso de Parámetros (cont.)

Ejemplo 6:
Se tiene el programa para cambiar el valor de dos variables (*swap*):

```
let a = 3
    b = 4
    swap = proc (x, y)
             let temp = x
             in begin
                  set x = y
                  set y = temp
                end
in begin
     (swap a b)
     -(a,b)
   end
```

Bajo el llamado por referencia, esto cambia los de valores de `a` y `b` retornando 1.
Si este programa fuera ejecutado con un interpretador con llamado por valor, retornaría -1, ya que la asignación dentro del procedimiento no tendría efecto sobre las variables `a` y `b`.

Paso de Parámetros (cont.)

En los llamados por referencia, los identificadores aún denotan referencias a valores expresados, luego los conjuntos no cambian:

$$\begin{array}{lcl} \text{Valor Denotado} & = & \text{Ref(Valor Expresado)} \\ \text{Valor Expresado} & = & \text{Número} + \text{ProcVal} \end{array}$$

El único cambio ocurre cuando se crean nuevas referencias. En los llamados por valor, una nueva referencia es creada para cada evaluación de un operando; en los llamados por referencia, una nueva referencia es creada para cada evaluación de un operando *distinto a una variable*.

Paso de Parámetros (cont.)

Para la implementación del llamado por referencia, una referencia será (como en llamado por valor) una referencia a una ubicación en un vector, pero dicho vector contendrá o los valores expresados o referencias a valores expresados (llamados *blanco directo* y *blanco indirecto* respectivamente).

Un blanco directo corresponde al comportamiento del llamado por valor, en el cual la nueva ubicación es creada; un blanco indirecto corresponde al nuevo comportamiento del llamado por referencia, en el cual no son creadas nuevas ubicaciones.

Un blanco es entonces un tipo de dato definido por

```
(define-datatype target target?
  (direct-target
    (expval expval?))
  (indirect-target
    (ref ref-to-direct-target?)))
```

Paso de Parámetros (cont.)

Los procedimientos que correspondientes a las variantes del tipo de dato `target` son:

```
(define expval?
  (lambda (x)
    (or (number? x) (procval? x))))

(define ref-to-direct-target?
  (lambda (x)
    (and
      (reference? x)
      (cases reference x
        (a-ref (pos vec)
          (cases target (vector-ref vec pos)
            (direct-target (v) #t)
            (indirect-target (v) #f)))))))
```

Paso de Parámetros (cont.)

Las nuevas definiciones de `deref` y `setref!` observan el tipo de blanco para determinar el valor expresado a retornar o la ubicación a cambiar

```
(define deref
  (lambda (ref)
    (cases target (primitive-deref ref)
      (direct-target (expval) expval)
      (indirect-target (ref1)
        (cases target (primitive-deref ref1)
          (direct-target (expval) expval)
          (indirect-target (p)
            (eopl:error 'deref
              "Illegal reference: ~s" ref1)))))))

(define setref!
  (lambda (ref expval)
    (let ((ref (cases target (primitive-deref ref)
                 (direct-target (expval1) ref)
                 (indirect-target (ref1) ref1))))
      (primitive-setref! ref (direct-target expval)))))
```

Paso de Parámetros (cont.)

El procedimiento `eval-expression` antes de la entrada del llamado por referencia era:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (varassign-exp (id rhs-exp)
        (begin
          (setref!
            (apply-env-ref env id)
            (eval-expression rhs-exp env))
          1))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args)))
      (if-exp (test-exp true-exp false-exp)
        (if (true-value? (eval-expression test-exp env))
          (eval-expression true-exp env)
          (eval-expression false-exp env)))
      ...
      )))
```

Paso de Parámetros (cont.)

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (proc-exp (ids body) (closure ids body env))
      (let-exp (ids rands body)
        (let ((args (eval-rands rands env)))
          (eval-expression body (extend-env ids args env))))
      (app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval? proc)
            (apply-procval proc args)
            (eopl:error 'eval-expression
              "Attempt to apply non-procedure ~s" proc))))
      (letrec-exp (proc-names idss bodies letrec-body)
        (eval-expression letrec-body
          (extend-env-recursively
            proc-names idss bodies env))))))
```

Paso de Parámetros (cont.)

Obeservando el código anterior, para implementar el llamado por referencia se necesitan hacer varios cambios.
Para aplicaciones primitivas, simplemente se necesita evaluar las subexpresiones y pasar los valores a `apply-primitive`, de manera que en `eval-expression` se escribe

```
(primapp-exp (prim rands)
  (let ((args (eval-primapp-exp-rands rands env)))
    (apply-primitive prim args)))
```

donde `eval-primapp-exp-rands` está definida por

```
(define eval-primapp-exp-rands
  (lambda (rands env)
    (map (lambda (x) (eval-expression x env)) rands)))
```

Paso de Parámetros (cont.)

Para ligadura local (formas `let`), se mantiene el comportamiento de los llamados por valor, luego en `eval-expression` se escribe

```
(let-exp (ids rands body)
  (let ((args (eval-let-exp-rands rands env)))
    (eval-expression body (extend-env ids args env))))
```

donde `eval-let-exp-rands` y `eval-let-exp-rand` están definidas por

```
(define eval-let-exp-rands
  (lambda (rands env)
    (map (lambda (x) (eval-let-exp-rand x env)) rands)))

(define eval-let-exp-rand
  (lambda (rand env)
    (direct-target (eval-expression rand env))))
```

Paso de Parámetros (cont.)

Para aplicación de procedimientos, se evalúa cada operando usando `eval-rand`.

Si el operando no es una variable, entonces se crea una nueva ubicación retornando el blanco directo. Si el operando es una variable, ésta denota una ubicación que contiene un valor expresado, luego se retorna un blanco indirecto que apunta a dicha ubicación.

Si una variable está ligada a una ubicación que contiene un blanco directo (el cual debe contener un valor expresado), entonces una referencia a la ubicación es retornada como un blanco indirecto. Pero si la variable está ligada a otra referencia, entonces dicha referencia es retornada.

```
(define eval-rand
  (lambda (rand env)
    (cases expression rand
      (var-exp (id)
        (indirect-target
          (let ((ref (apply-env-ref env id)))
            (cases target (primitive-deref ref)
              (direct-target (expval) ref)
              (indirect-target (ref1) ref1)))))
      (else
        (direct-target (eval-expression rand env))))))
```

Paso de Parámetros (cont.)

Ejemplo 7:

```
(proc (t, u, v, w)
   (proc (a, b)
      (proc (x, y, z)
         set y = 13
       a b 6)
    3 v)
 5 6 7 8)
```

Este ejemplo puede verse gráficamente como la operación de `eval-rand` de la siguiente figura.