Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

•••

Karla Ramírez Pulido

Recursión
Parte II

Definimos tipos de datos

Valores

```
(define-type RCFAE-Value
        [numV (n number?)]
        [closureV  (param symbol?)
                   (body RCFAE?)
                   (env Env?)])
```

Definimos tipos de datos

Ambientes

```
(define-type Env
    [mtSub]
    [aSub(name symbol?)
        (value RCFAE-Value?)
        (env Env?)]
    [aRecSub (name symbol?)
             (value boxed-RCFAE-Value?)
             (env Env?)])
```

Introduciendo CAJAS

Cajas: estructuras de datos que pueden almacenar un único elemento

- box
- unbox
- box?

```
(define (boxed-RCFAE-Value? v)
        (and   (box? v)
               (RCFAE-Value? (unbox v))))
```

Función lookup

Busca un símbolo en el ambiente

;; lookup : symbol env→RCFAE-Value

```
(define (lookup name env)
   (type-case Env env
      [mtSub() (error'lookup "no binding for
      identifier")]
      [aSub(bound-name bound-value rest-env)
         (if (symbol=? bound-name name)
             bound-value
             (lookup name rest-env))]
      [aRecSub(bound-name boxed-bound-value rest-env)
          (if (symbol=? bound-name name)
              (unbox boxed-bound-value)
              (lookup name rest-env))]))
```

```
;; cyclically-bind-and-interp : symbolRCFAEenv→env

(define (cyclically-bind-and-interp bound-id  named-expr  env)
    (local ([define value-holder (box (numV 1729))]
                [define  new-env (aRecSub  bound-id  value-holder  env)]
                [define  named-expr-val (interp   named-expr   new-env)])
                    (begin
                        (set-box!  value-holder  named-expr-val)
                        new-env)))
```

Recursión: Intérprete

;; interp : RCFAE env→RCFAE-Value

```
(define (interp expr env)
    (type-case RCFAE expr
        [num (n) (numV n)]
        [add (l r) (num+ (interp l env) (interp r env))]
        [mult (l r) (num*(interp l env) (interp r env))]
        [if0 (test-expr  then-expr  else-expr)
             (if (num-zero? (interp test-expr env))
                 (interp then-expr env)
                 (interp else-expr env))]
```

Recursión: Intérprete (sig 2)

;; interp : RCFAE env→RCFAE-Value

```
[id (v) (lookup v env)]

[fun (bound-id bound-body)

     (closureV bound-id bound-body env)]

[app (fun-expr  arg-expr)

     (local ([define fun-val (interp fun-expr env)])

          (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

                                                (interp arg-expr env)

                                                (closureV-env fun-val))))]
```

Recursión: Intérprete (sig 3)

;; interp : RCFAE env→RCFAE-Value

```
[rec (bound-id named-expr bound-body)
     (interp bound-body
             (cyclically-bind-and-interp bound-id  named-expr  env))]))
```

Ejecución de interp: números y sumas

> (interp (numV 3) ())

(numV 3)

> (interp (add (numV 3) (numV 2)) ())

Sumar el interp del lado izq y el interp del lado der.

(num+ (interp (numV 3) ()) (interp (numV 2) ()))

(+ 3 2) => 5 => (numV 5)

```
(define (interp expr env)
  (type-case RCFAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env)
                     (interp r env))]
```

Ejecución de interp: ids y funciones

> (interp (id z) ( ))

(lookup (id z) ( ))

(error lookup ”no binding for identifier”)

```
(define (interp expr env)

  (type-case RCFAE expr

                …

   [id (v) (lookup v env)]  …))
```

```
(define (lookup name env)

  (type-case Env env

    [mtSub() (error’lookup ”no binding for
    identifier”)]

    [aSub(bound-name bound-value rest-env)

       (if (symbol=? bound-name name) bound-value

          (lookup name rest-env))]
```

```
[aRecSub(bound-name boxed-bound-value
rest-env)

   (if (symbol=? bound-name name)

         (unbox boxed-bound-value)

         (lookup name rest-env))]))
```

Ejecución de interp: funciones

```
(define (interp expr env)
  (type-case RCFAE expr
    …
    [fun (bound-id bound-body)
         (closureV bound-id
                   bound-body
                   env)]
```

```
> (interp (fun (id x) (id x)) ( ) )
  (closureV (id x) (id x) ( ) )
```

Ejecución de interp: if0

> (interp (if0 (numV 5) (numV 1) (numV 2)) ( ))

(**if** (num-zero? (interp (numV 5) ( )))

(num-zero? (numV 5)) NOOO

(**if** false

interpreto la expresión else-expr

i.e. (interp (numV 2) ( ))

= (numV 2)

```
(define (interp expr env)
…
 [if0 (test-expr  then-expr  else-expr)
      (if (num-zero? (interp test-expr env))
                     (interp then-expr env)
                     (interp else-expr env))]
```

Ejecución de interp: app y rec

> (interp {rec {{id fac} {fun {n}

{if0 n 1 {* n {fac {-n 1} } } } }

{fac 1} } ())

Recordemos:

{fac 1} = (1 (fac 0))*

*= (* 1 1) = 1*

Ejecución de interp: app y rec

> (interp {rec {{id fac} {fun {n}

{if0 n 1 {* n {fac {-n 1}}}}}}

{fac 1} } ())

```
(define (interp expr env)

 [rec (bound-id named-expr bound-body)

          (interp bound-body

                        (cyclically-bind-and-interp bound-id  named-expr  env))]))
```

Ejecución de interp: app y rec

> (interp {rec { {id fac} {fun {n}

{if0 n 1 {* n {fac {-n 1}}}}} }

{fac 1} } ())

```
[rec (bound-id named-expr bound-body)
        (interp bound-body
                (cyclically-bind-and-interp bound-id  named-expr  env))]))
```

Ejecución de interp: app y rec

> (interp {rec { {id fac} {fun {n}

{if0 n 1 {* n {fac {-n 1}}}}} }

{fac 1} } ())

```
[rec (fac {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  {fac 1})

        (interp bound-body

                (cyclically-bind-and-interp bound-id  named-expr  env))]
```

Ejecución de interp: app y rec

> (interp {rec { {id fac} {fun {n}

{if0 n 1 {* n {fac {-n 1}}}}} }

{fac 1} } ())

```
[rec (fac {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  {fac 1})
         (interp {fac 1}
                 (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
         )]
```

Ejecución de interp: app y rec

```
> (interp {fac 1}

          (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
```

Interp recibe: una expresión y un ambiente

La expresión es una app de función {fac 1}

Y el ambiente es el vacío = ()

Para hacer la aplicación de función {fac 1} necesitamos resolver primero la llamada de la función: cyclically-bind-and-interp

Ejecución de interp: cyclically-bind-and-interp

```
(cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
```

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    (begin
      (set-box! value-holder named-expr-val)
      new-env)))
```

Ejecución de interp: cyclically-bind-and-interp

```
1  (define (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
2    (local  ( [define value-holder (box (numV 1))]                          value-holder
3              [define  new-env (aRecSub  bound-id  value-holder  env)]
4              [define  named-expr-val (interp   named-expr   new-env)])
5                 (begin
6                       (set-box!  value-holder  named-expr-val)
7                       new-env)))
```

Ejecución de interp: cyclically-bind-and-interp

1 (define (**cyclically-bind-and-interp** `fac` `{fun {n} {if0 n 1 {* n {fac {-n 1}}}}}` `()`)

2 (local ([define value-holder (box (numV 1))]

3 [define new-env (aRecSub bound-id value-holder ())]

value-holder

new-env = (aRecSub `fac` value-holder ())

Recordemos cómo se define un Ambiente

mtSub= ( )

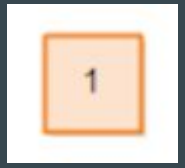
aSub = ((aSub x 5))

aRecSub= ((aRecSub fac def-función))

```
(define-type Env
      [mtSub]
      [aSub(name symbol?)
            (value RCFAE-Value?)
            (env Env?)]
      [aRecSub (name symbol?)
               (value
          boxed-RCFAE-Value?)
               (env Env?)])
```

Ejecución de interp: cyclically-bind-and-interp

```
1  (define (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  () )
2    (local  ( [define value-holder (box (numV 1))]
3              [define  new-env (aRecSub  fac  value-holder  ())]
4              [define  named-expr-val (interp  named-expr  new-env)])
```



value-holder

named-expr-val = (interp {fun {n} {if0 n 1 {* n {fac {-n 1}}}}} new-env)

named-expr-val = (interp {fun {n} {if0 n 1 {* n {fac {-n 1}}}}} (aRecSub fac value-holder ()))

named-expr-val = (closureV n {if0 n 1 {* n {fac {-n 1}}}} (aRecSub fac value-holder ()))

Ejecución de interp: cyclically-bind-and-interp

```
1  (define (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
2    (local  ( [value-holder = (box (numV 1))]
3              [new-env = (aRecSub  fac  value-holder  ())]
4              [named-expr-val  = (closureV  n  {if0 n 1 {* n {fac {-n 1}}}}
                                                (aRecSub  fac  value-holder  ()) ) ])
5                 (begin
6                       (set-box!  value-holder  named-expr-val)
7                       new-env)))
```

value-holder

Ejecución de interp: cyclically-bind-and-interp

1 (define (**cyclically-bind-and-interp** `fac` `{fun {n} {if0 n 1 {* n {fac {-n 1}}}}}` **(**`()`**)**

2 value-holder = (box (numV 1))

3 new-env = **(aRecSub fac value-holder ())**

4 named-expr-val = (closureV n {if0 n 1 {* n {fac {-n 1}}}}

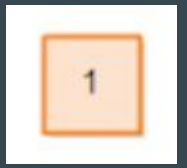
(aRecSub fac value-holder ()))

5 (begin

6 (set-box! value-holder named-expr-val)

i.e (set-box! value-holder (closureV n {if0 n 1 {* n {fac {-n 1}}}}

(aRecSub fac value-holder ()))



value-holder

Ejecución de interp: cyclically-bind-and-interp

1 (define (**cyclically-bind-and-interp** **fac** {fun {n} {if0 n 1 {* n {fac {-n 1}}}}} **()**)

2 value-holder = (box (numV 1))

3 new-env = **(aRecSub fac value-holder ())**

4 named-expr-val = (closureV n {if0 n 1 {* n {fac {-n 1}}}}

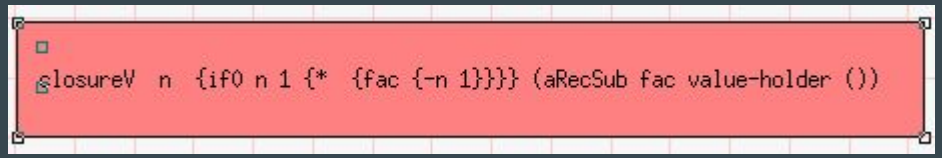
(aRecSub fac value-holder ()))

5

i.e (set-box! value-holder (closureV n {if0 n 1 {* n {fac {-n 1}}}}

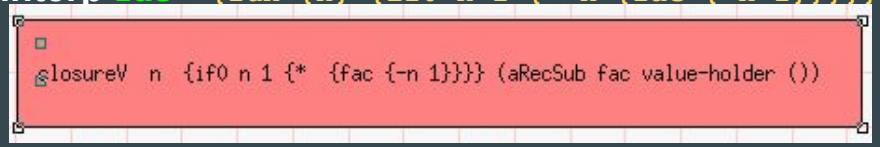
(aRecSub fac value-holder ()))

Ahora tenemos

value-holder = 

Recordemos que teníamos, solo nos falta la línea 7

```
1  (define (cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
2    (local  ( [value-holder =
3              [new-env = (aRecSub  fac  value-holder  ())]
4              [named-expr-val  = (closureV  n  {if0 n 1 {* n {fac {-n 1}}}}
                                                  (aRecSub  fac  value-holder  ()) ) ])
5                  (begin
6                        (set-box!  value-holder  named-expr-val)
7                        new-env)))
```

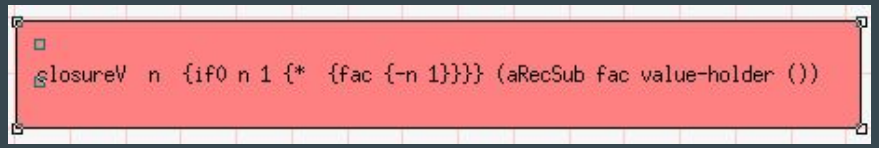


Recordemos que teníamos, solo nos falta la línea 7

Entonces **cyclically-bind-and-interp** regresa un ambiente= new-env

new-env = (aRecSub `fac` value-holder ())

value-holder =



Y todo esto para regresar a interp

```
(interp {fac 1}
          (cyclically-bind-and-interp fac
                {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
```

Ejecución de interp: app y rec

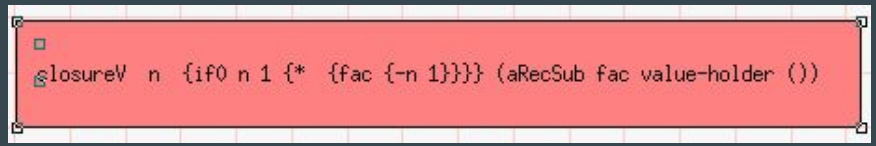
```
> (interp {fac 1})
```

```
(cyclically-bind-and-interp fac  {fun {n} {if0 n 1 {* n {fac {-n 1}}}}}  ())
```

= new-env = (aRecSub `fac` value-holder ())

y

value-holder =



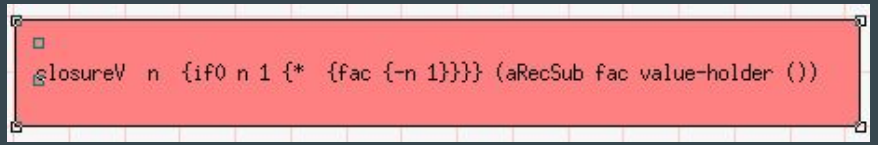
Ejecución de interp: app y rec

```
> (interp {fac 1}
```

new-env = (aRecSub `fac` value-holder ())

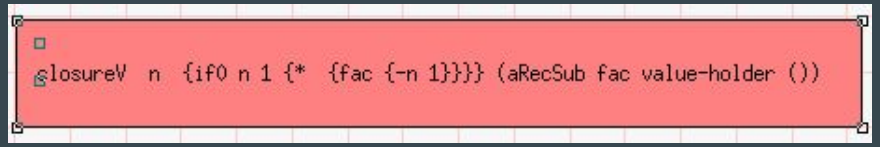
Y esto es la evaluación de una aplicación de función en un ambiente (new-env)

value-holder =



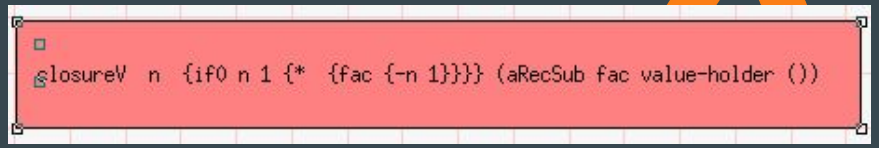
Ambiente (Env) solo tiene un registro de tipo aRecSub

value-holder=



Ambiente, el cual extendió del amb. vacío

Ejecución de interp: app y rec




```
> (interp {fac 1}
```

new-env = (aRecSub `fac` value-holder ())

En interp:

```
[app (fun-expr  arg-expr)
     (local ([define fun-val (interp fun-expr env)])
          (interp (closureV-body fun-val) (aSub (closureV-param fun-val)
                                                (interp arg-expr env)
                                                (closureV-env fun-val))))]
```

```
(interp {fac 1}
        (aRecSub fac value-holder ()))
```

[app (fac 1)

fun-expr = fac

arg-expr = 1

```
[app (fun-expr arg-expr)
(local([define fun-val (interp fun-expr env)])
  (interp (closureV-body fun-val) (aSub (closureV-param  fun-val)
                                        (interp arg-expr env)
                                        (closureV-env fun-val))))]
```

```
(interp {fac 1}
        (aRecSub fac value-holder ()))
```

```
closureV  n  {if0 n 1 {*  {fac {-n 1}}}} (aRecSub fac value-holder ())
```

1 [app (fac 1)

2 fun-val = (interp fac (aRecSub fac value-holder ()))

```
1  [app (fun-expr arg-expr)
2   (local([define fun-val (interp fun-expr env)])
3       (interp (closureV-body fun-val) (aSub (closureV-param
fun-val)
4                                       (interp arg-expr env)
5                                       (closureV-env fun-val))))]
```

Evaluando la aplicación de función: {fac 1}

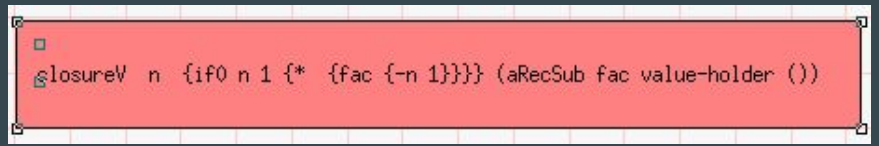
```
(interp {fac 1}  (aRecSub fac value-holder ())
```

[app (fac 1)

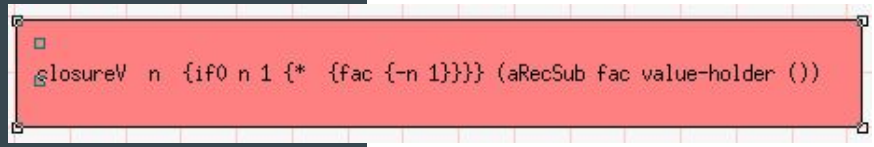
```
fun-val = (interp fac (aRecSub fac value-holder ()) )
```

evaluar un id = fac se hace llamando a la función lookup en un ambiente

value-holder =




```
fun-val = (closureV  n  {if0 n 1 {* n {fac {- n 1}}}}
                        (aRecSub fac value-holder ()))
```



```
(interp {fac 1}
        (aRecSub fac value-holder ())
```

```
1 [app ( fac 1)
2   fun-val =
(closureV  n  {if0 n 1 {* … }}}}
       (aRecSub fac value-holder ()))
```

```
1 [app (fun-expr arg-expr)
2  (local([define fun-val (interp fun-expr env)])
3    (interp (closureV-body fun-val)
                    (aSub (closureV-param  fun-val)
4                   (interp arg-expr env)
5                   (closureV-env fun-val))))]
```

Siguiendo la evaluación: línea 3

Para poder evaluar la línea 3 (interp) primero tenemos que evaluar las sub-expresiones:

3 (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

4 (interp arg-expr env)

5 (closureV-env fun-val))))]

(closureV-body fun-val)

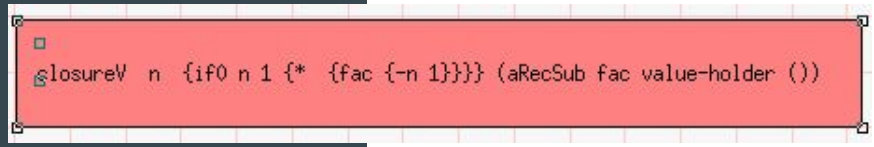
fun-val = (closureV

n

{if0 n 1 {* … }}}}

(aRecSub fac value-holder ()))

= {if0 n 1 {* n {fac {- n 1}}}}



```
(interp {fac 1}
        (aRecSub fac value-holder ())
```

```
1 [app ( fac 1)
2   fun-val =
(closureV  n  {if0 n 1 {* … }}}}
       (aRecSub fac value-holder ()))
```

```
1 [app (fun-expr arg-expr)
2  (local([define fun-val (interp fun-expr env)])
3    (interp (closureV-body fun-val)
                    (aSub (closureV-param  fun-val)
4                   (interp arg-expr env)
5                   (closureV-env fun-val))))]
```

Siguiendo la evaluación: línea 3

Para poder evaluar la línea 3 (interp) primero tenemos que evaluar las sub-expresiones:

3 (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

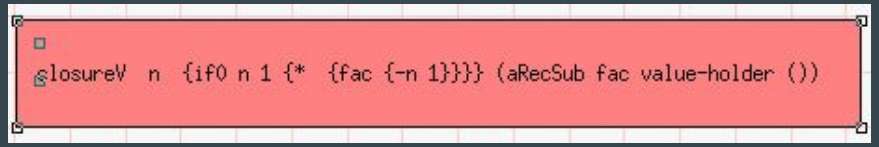
(**closureV-param** fun-val)

4 (interp arg-expr env)

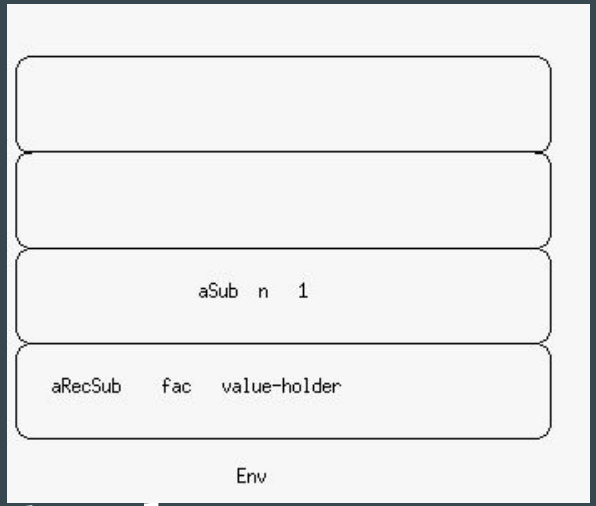
5 (closureV-env fun-val))))]

```
fun-val = (closureV  n
                {if0 n 1 {* … }}}}
          (aRecSub fac value-holder ()))
```

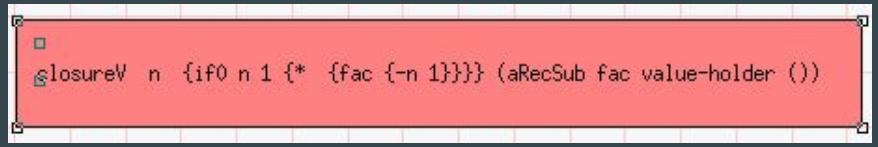
= **n**



Esto es lo que va a pasar: Ambiente se extiende con la asignación n = 1



value-holder=



Ambiente extendido con n=1

Siguiendo la evaluación: línea 4

Para poder evaluar la línea 3 (interp) primero tenemos que evaluar las sub-expresiones:

(interp arg-expr env)

3 (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

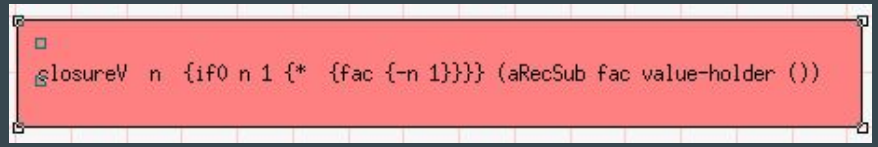
4 (interp arg-expr env)

5 (closureV-env fun-val))))]

[app (fac 1)

fun-expr = fac

arg-expr = 1



= 1

Siguiendo la evaluación: línea 5

Para poder evaluar la línea 3 (interp) primero tenemos que evaluar las sub-expresiones:

3 (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

(closureV-env fun-val)

4 (interp arg-expr env)

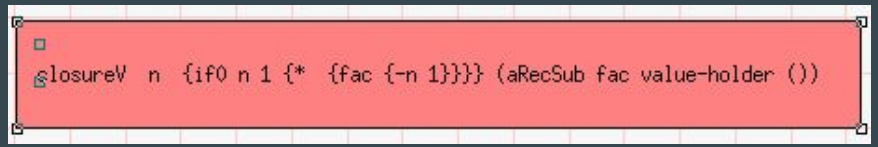
fun-val = (closureV n

5 (closureV-env fun-val))))]

{if0 n 1 {* … }}}}

(aRecSub fac value-holder ()))

= (aRecSub fac value-holder ())



Siguiendo la evaluación: línea 3

Sustituyendo:

(interp

{if0 n 1 {* n {fac {- n 1}}}}

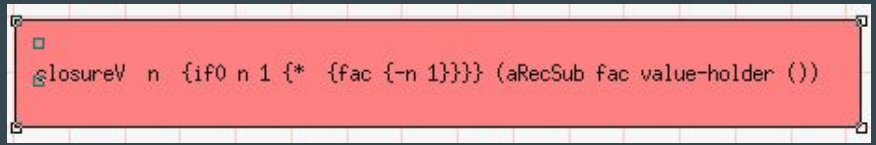
(aSub n 1 (aRecSub fac value-holder ()))

)

3 (interp (closureV-body fun-val) (aSub (closureV-param fun-val)

4 (interp arg-expr env)

5 (closureV-env fun-val))))]



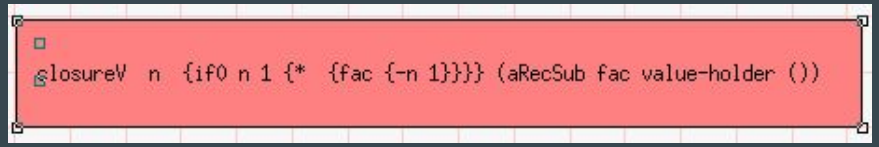
Siguiendo la evaluación: del if0

(interp

{if0 n 1 {* n {fac {- n 1}}}} **(aSub n 1 (aRecSub fac value-holder ())))**

If0 evalúa su expresión condicional, i.e. n

n a su vez es un identificador, entonces lo busca (lookup) en el ambiente y como lo encuentra regresa su valor que es 1



Siguiendo la evaluación: del if0

(interp

{if0 n 1 {* n {fac {- n 1}}}} **(aSub n 1 (aRecSub fac value-holder ())))**

Sustituimos n por su valor 1 en la expresión

(interp

{if0 1 1 {* 1 {fac {- 1 1}}}} **(aSub n 1 (aRecSub fac value-holder ())))**

Evalúamos la condicional del if0: (iszero? 1) → FALSE entonces evalúamos la rama del else del if0, i.e. **{* 1 {fac {- 1 1} }}**

Siguiendo la evaluación: del if0

(interp

{* 1 {fac {- 1 1}}} (aSub n 1 (aRecSub fac value-holder ())))

Evalúamos la expresión multiplicación: {* 1 {fac {-1 1}}}

- Evaluamos el lado izquierdo: es un número entonces regresamos el mismo número = 1

  (interp 1 env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

- Evaluamos el lado derecho: es una aplicación de función = {fac {-1 1}}

  (interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

  IMPORTANTE: la multiplicación tiene que esperar a evaluarse pues su lado derecho es otra expresión (app).

Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp

{* 1 {fac {- 1 1}}} (aSub n 1 (aRecSub fac value-holder ())))

- Antes de hacer la multiplicación evalúaremos el lado derecho: {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

[app (fun-expr arg-expr)

(local ([define fun-val (interp fun-expr env)])

(interp (closureV-body fun-val) (aSub (closureV-param fun-val) (interp arg-expr env)

(closureV-env fun-val))))]

Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

[app (fun-expr arg-expr)

(local ([define fun-val (interp fun-expr **(aSub n 1 (aRecSub fac value-holder ()))**)])

fun-val = (interp fac **(aSub n 1 (aRecSub fac value-holder ())))**

Evaluar un id es buscarlo en el ambiente, ¿lo encuentra? SÍ

entonces obtiene su valor =

```
(closureV  n  {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))
```

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

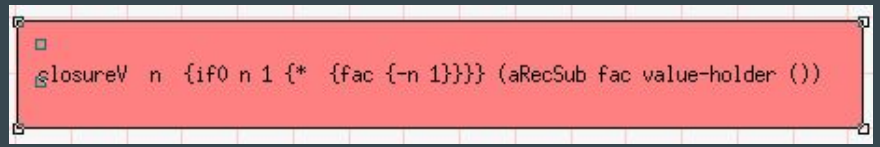
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp (closureV-body fun-val) (aSub (closureV-param fun-val)

(interp arg-expr env)

(closureV-env fun-val))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

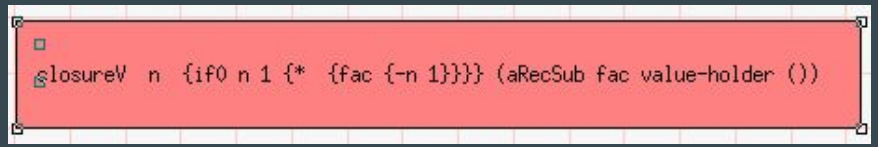
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **(closureV-body fun-val)** (aSub (closureV-param fun-val)

(interp arg-expr env)

(closureV-env fun-val))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

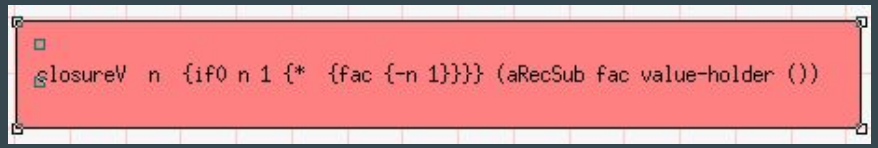
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub (closureV-param fun-val)

(interp arg-expr env)

(closureV-env fun-val)))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

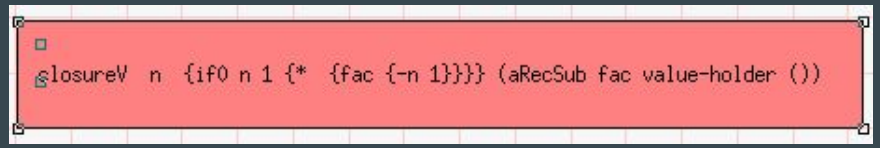
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **(closureV-param fun-val)**

(interp arg-expr env)

(closureV-env fun-val))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

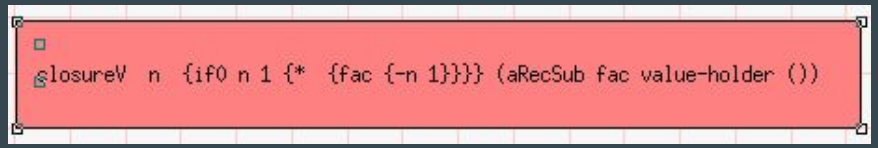
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

(interp arg-expr env)

(closureV-env fun-val)))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

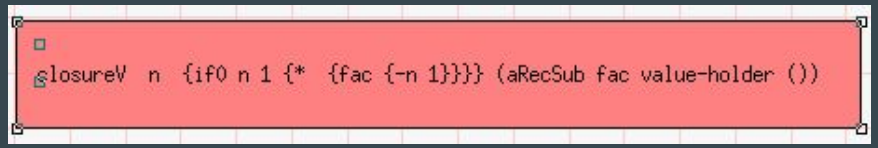
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

(interp arg-expr env)

(closureV-env fun-val)))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

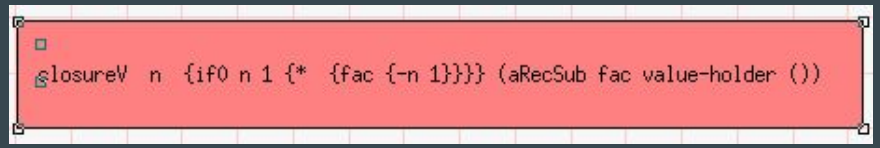
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

(interp {- 1 1} env)

(closureV-env fun-val)))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

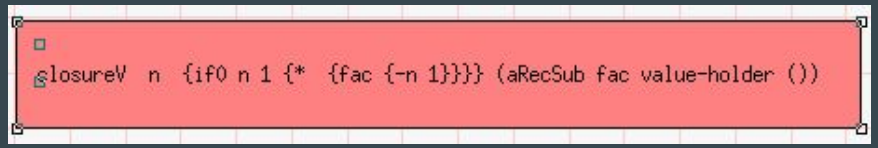
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

(interp (- (interp 1 env) (interp 1 env))

(closureV-env fun-val)))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

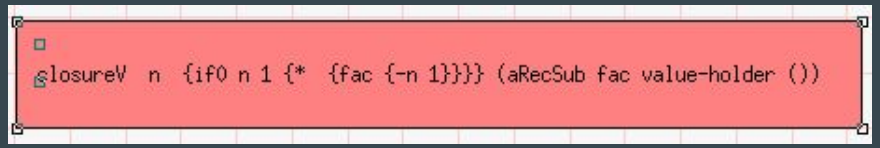
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

0

(closureV-env fun-val))))]



Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

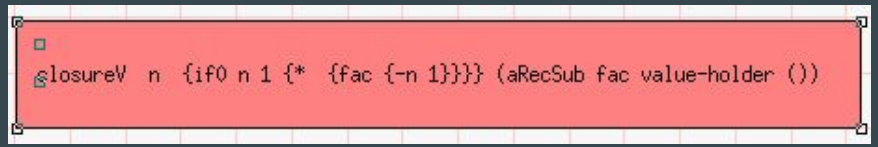
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

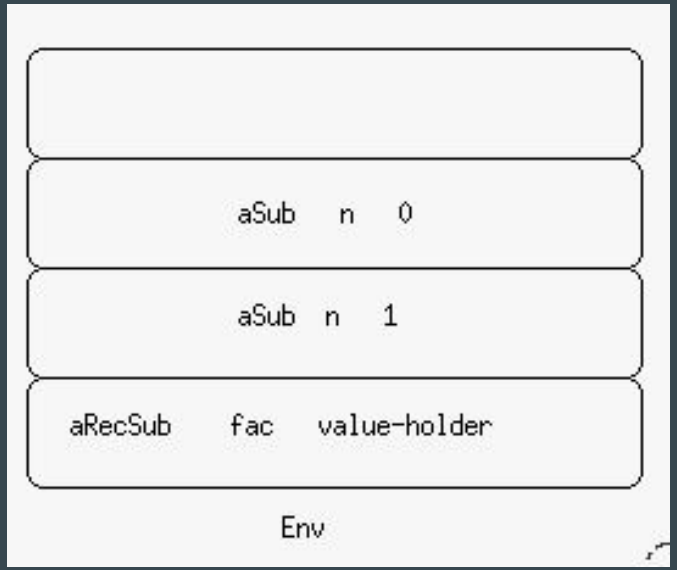
(interp **{if0 n 1 {* … }}}}** (aSub **n**

0

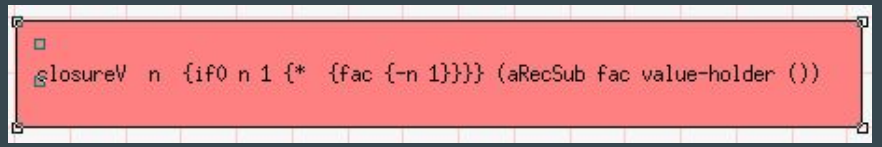
(closureV-env fun-val)))))]



Ambiente se extiende con la asignación n = 0



value-holder=



Ambiente con aSub n=0 y extendió del ambiente anterior

Siguiendo la evaluación de la aplicación {fac {-1 1}}

(interp {fac {- 1 1}} env) donde env = **(aSub n 1 (aRecSub fac value-holder ()))**

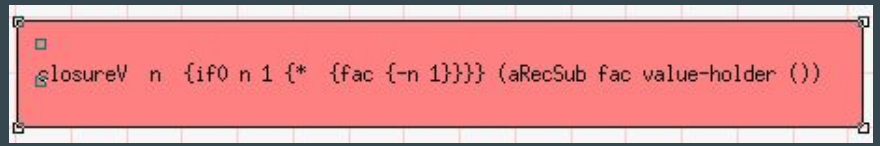
[app (fun-expr arg-expr)

fun-val = (closureV n {if0 n 1 {* … }}}} (aRecSub fac value-holder ()))

(interp **{if0 n 1 {* … }}}}** (aSub **n**

0

(aRecSub fac value-holder ()))))]



Ahora evaluamos la expresión if0

> (interp **{if0 n 1 {* … }}}}** (aSub **n 0 (aRecSub fac value-holder ())**)))]

Susituyendo n por su valor = 0

(interp **{if0 0 1 {* … }}}}** (aSub **n 0 (aRecSub fac value-holder ())**)))]

Evaluando la condicional del if0 (iszero? 0) TRUE

i.e. regresamos el valor de 1

Ahora si estamos listos para evaluar la MULTIPLICACIÓN:

{* 1 {fac 0}} = {* 1 1}

Evaluando la multiplicación: {* 1 1}

Evaluar el lado izquierdo y después el lado derecho

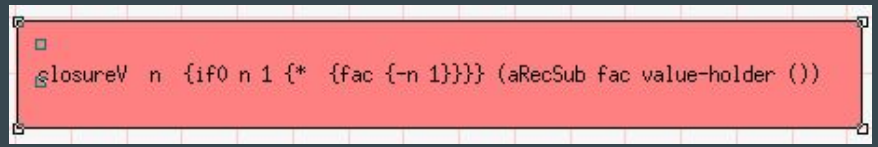
(interp

{* 1 1} (aSub n 0 (aSub n 1 (aRecSub fac value-holder ()))))

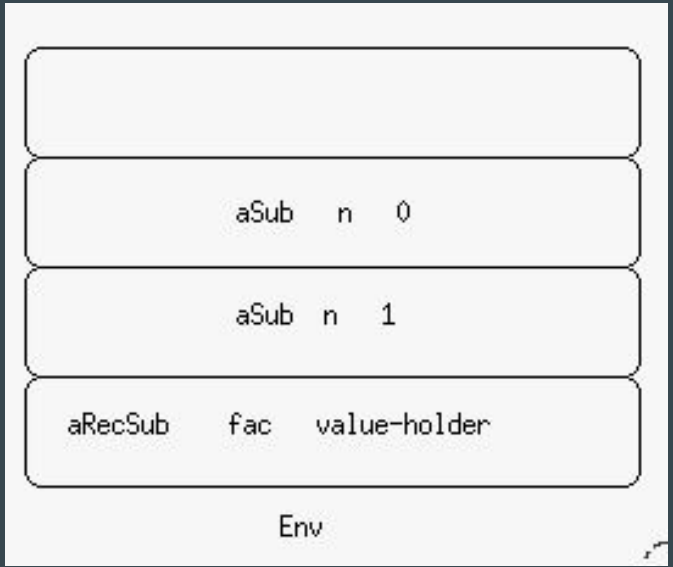
lado izq: (interp 1 env) = 1

lado der: (interp 1 env) = 1

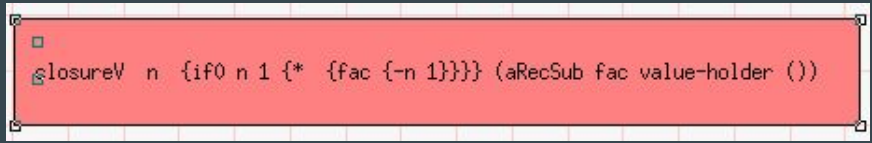
(* 1 1) = 1



Ambiente final



value-holder=



La caja value-holder mantiene en todo momento la cerradura que se crea cuando existe una función (en este caso recursiva). La caja contiene dentro la cerradura (closureV) con el nombre de la función fac, el cuerpo de la función de factorial y el ambiente del cual extendió.

¿Alguna duda?

Gracias