PROC

Fermin Alejandro Ahumada Garcia

13 de octubre de 2022

Resumen

Ejercicios

1. Problema 3.19

Sintaxís Concreta

```
Expresion:= letproc-exp(fun param body exp)
```

Semántica

```
(value-of(letproc-exp fun param body exp)&)
=
(value-of exp1[name=(proc-val(procedure param body ))]&)
```

2. Problema 3.20

In PROC, procedures have only one argument, but one can get the effect of multiple argument procedures by using procedures that return other procedures. For example, one might write code like:

```
    let f = proc (x) proc (y) ...
    in ((f 3) 4)
```

Para poder ejecutarlo lo corremos como:

```
(run '(let f= proc(x) proc (y) -(x,-(0,y))))
```

Siendo un proces curry donde al tomar dos argumentos se le devuelve la suma.Que nosotros representamos esto como -(x,-(0,y)). Al proc solo tener dos cosas , su variable y un cuerpo ,el enunciado nos da dos argumentos , al ser dos tenemos que dejar la variable de la primera y en el cuerpo , metemos otro Proc para así poder hacer la suma del proceso.

Así dando que la variable "x"sea asignada desde el primero Proc y su cuerpo,seria el return de z",consiguiendo la suma a travez de Proc(y),haciendo valida la interpretación.

3. Problema 3.27

Add a new kind of procedure called a traceproc to the language. A traceproc works exactly like a proc, except that it prints a trace message on entry and on exit.

En esto tenemos que plantear un proceso Proc , donde este recibe:

Proc-val= Int-Bool-Proc

```
#Sintaxis Concreta
Expression:= traceproc(Identifier) Expression
#Sintaxis Abstracta
(traceproc-exp var body)
```

```
#Semantica
(value-of(traceproc-exp(var body)env))
=
(proc-val (procedure var body #t)env)
```

4. Problema 3.25

The tricks of the previous exercises can be generalized to show that we can define any recursive procedure in PROC. Consider the following bit of code:

```
let makerec = proc (f)
    let d = proc (x)
        proc (z) ((f (x x)) z)
    in proc (n) ((f (d d)) n)
in let maketimes4 = proc (f)
        proc (x)
        if zero?(x)
        then 0
        else -((f -(x,1)), -4)
in let times4 = (makerec maketimes4)
    in (times4 3)
```

Show that it returns 12.

maketimes4 es un procedimiento que toma un procedimiento times4 y devuelve este mismo.Lo que se hace es tomar el maketimes4 y se hace un procedimiento maker que toma un maker y devuelve un procedimiento times4(contador).

```
let makerec = proc (f)
    let maker = proc (maker)
        let recursive-proc =(maker maker)
    in proc ((f recursive-proc) n)
    in (maker maker)
```

Lo que hace que funcione el codigo es (maker maker),dando que no se pueda llamar asi nomas , por eso metemos en otro procedimiento para arreglar esto, dando que pueda salir de la recursividad infinita. Co lo que ya separado nos daría 12.

5. Problema 3.29

Unfortunately, programs that use dynamic binding may be exceptionally difficult to understand. For example, under lexical binding, consistently renaming the bound variables of a procedure can never change the behavior of a program: we can even remove all variables and replace them by their lexical addresses, as in section 3.6. But under dynamic binding, this transformation is unsafe. For example, under dynamic binding, the procedure proc (z) a returns the value of the variable a in its caller's environment. Thus, the program

```
    let a = 3
    in let p = proc (z) a
        in let f = proc (x) (p 0)
```

```
4               in let a = 5
5                   in (f 2)
```

returns 5, since a's value at the call site is 5. What if f's formal parameter were a?

Si retomamos que es "formal parameter",se refiere a un error en los parametros , en este caso seria en la variable var , que se encuentra en proc-exp (var body). La mejor opcion es cambiar el parametro de f en el codigo,dando:

```
1       let a=3
2       in let p = proc (z) a
3           in let f = proc (a) (p 0)
4               in let a = 5
5                   in (f 2)
```

Donde esto asignara el valor de 2 a la variable (a) cuando se llame al procedimiento f.