Control 2 – Lenguajes de Programación

Departamento de Ciencias de la Computación - Universidad de Chile

Profesor: Tomás Barros

8 de octubre de 2009

1. Pregunta 1

1. Se propone la siguiente implementación de la función `interp`:

```
(define (interp expr ds)
  (type-case FAE expr
    ...
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr ds)])
                (interp (closureV-body fun-val)
                        (aSub (closureV-param fun-val)
                              (interp arg-expr ds)
                              ds))))]))
```

Explique si la anterior implementación cumple con la definición de scope dinámico o estático o ninguno de los dos anteriores. Provea ejemplos para justificar su respuesta.

Respuesta: Implementa scope dinámico en la aplicación de funciones ya que el cuerpo de la función, que pudo ser definida en un scope sintáctico distinto, recibe el ambiente actual y por ende las variables del ambiente al momento de aplicar la función ingresan al cuerpo de ésta. Por ejemplo:

```
{with {x 4}
      {with {f {fun {y} {+ x y}}}
            {with {x 5} {f 10}}}}
```

Retorna 15 en vez de 14 debido a que toma el valor de x=5 en la aplicación y no de x=4 de su definición, como debiese ser si estuviesemos en scope estático.

La versión correcta es:

```
(define (interp expr ds)
  (type-case FAE expr
    ...
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr ds)])
                (interp (closureV-body fun-val)
                        (aSub (closureV-param fun-val)
                              (interp arg-expr ds)
                              (closureV-ds fun-val))))]))
```

2. Suponga la interpretación del siguiente código bajo el lenguaje RCFAE visto en clase (evaluación eager, scope estático):

```
{rec {fact {fun {x}
             {if0 x
                  1
             {* {fact {- x 1}} x}}}}
   {fact 3}}
```

- ¿Que resultado se obtiene de la evaluación?
  Respuesta: Obtiene 6
- ¿Cambia el resultado ahora si el intérprete implementa scope dinámico?, explique.
  Respuesta: No cambia, al menos no en el intérprete RCFAE visto en clases. Esto es debido a que si bien estamos en un ambiente dinámico, RCFAE toma el último valor definido para x (que se crea cada vez que se aplica la función) (al momento de la última llamada a fact, el ambiente es `x ->3, x ->2, x->1`).
  Visto de otro puento de vista, en este problema particular el scope estático de x coincide con el scope dinámico de x por lo que en ambos casos el resultado es el mismo.

2. Pregunta 2

Al definir la operación de substitución, uno de los problemas principales fue especificar bien lo que pasa cuando se usa el mismo identificador varias veces. Nicolas De Bruijn propuso reemplazar nombres por números, más bien, *índices*, que indican la profundidad de una asociación (binding).[1]

Por ejemplo, en vez de escribir: `{with {x 5}{+ x x}}`, podemos escribir: `{with 5 {+ <0> <0>}}`. Por convención el alcance (scope) corriente es `0`, entonces `x` se reemplazó por el índice `<0>`. Con esta representación, basta saber que la presencia de un `with` indica que entramos en un nuevo alcance. Así mismo, la siguiente expresión y su equivalente:

```
{with {x 5}                {with 5
  {with {y 4}                {with 4
    {+ x y}}}                  {+ <1> <0>}}}
```

1. Escriba la conversión de la siguiente expresión usando índices de De Bruijn:

```
{with {x 5}
  {with {y {+ x 2}}
    {+ x
       {with {y {- y 1}}
         {+ y x}}}}}
```

Respuesta:

```
{with 5
  {with {+ <0> 2}
    {+ <1>
       {with {- <0> 1}
         {+ <0> <2>}}}}}
```

2. Extienda la gramática del lenguaje `WAE` para acomodar expresiones con índices de De Bruijn.

Respuesta:

```
(define-type WAE
   [num (n number?)]
   [bj (n number?)]
   [add (l WAE?)
        (r WAE?)]
   [sub (l WAE?)
        (r WAE?)]
  [with (name symbol?)(named-expr WAE?)(body WAE?)]
  [withbj (named-expr WAE?)(body WAE?)]
  [id (name symbol?)])
```

[1] Esa representación es usada por (casi) todos los compiladores.

3. Implemente la función `toDeBruijn :: WAE ->WAE` que transforma una expresión con `with` normales a una expresión con `with` modificado e índices De Bruijn.

Respuesta:

```
(define-type MAP
  [mtMap]
  [aMap (x symbol?) (next MAP?)]
  )

(define (lookup name mymap level)
  (type-case MAP mymap
             [mtMap () (error 'env-lookup "no binding for identifier")]
             [aMap (x rest-map)
                   (if (symbol=? x name)
                       level
                       (lookup name rest-map (+ 1 level)))]))

(define (toDeBruijnAux sexp mymap)
  (type-case WAE sexp
             [num (n) sexp]
             [bj (n) sexp]
             [add (l r) (add (toDeBruijnAux l mymap) (toDeBruijnAux r mymap))]
             [sub (l r) (sub (toDeBruijnAux l mymap) (toDeBruijnAux r mymap))]
             [with (bound-id named-expr bound-body)
                   (withbj (toDeBruijnAux named-expr mymap)
                           (toDeBruijnAux bound-body (aMap bound-id mymap)))]
             [id (v) (bj (lookup v mymap 0))]
             [withbj (bound-id named-expr) sexp]
             ))


(define (toDeBruijn sexp)
  (toDeBruijnAux sexp (mtMap)))
```

3. Pregunta 3

1. Implemente en Scheme la función `if0` que toma tres parámetros, `e1`, `e2` y `e3`, tal que evalua `e2` si `e1` vale 0, `e3` sino. ¿Por qué no es posible, en Scheme, definir `if0` para que se use de la siguiente manera?:

```
(if0 (- 4 5)
     (write "zero")
     (write "not zero")))  --> "not zero"
```

   Cambie el programa anterior para que funcione con su definición de `if0`.

   Respuesta: No es posible porque `if0` es una función que recibe tres argumentos, como scheme es eager, esos 3 argumentos son evaluados al aplicar la función y por ende se ejecutaría ambos write. La solución es encapsular en funciones anónimas, la función es evaluada (pero no aplicada) al momento de aplicar la función if0, y dependiendo de si es verdadero o falso se aplicará la función correcta.

```
(define (if0 e1 e2 e3)
(if (= 0 e1) (e2) (e3)))

(if0 (- 4 5)
     (lambda () (write "zero"))
     (lambda () (write "not zero")))
```

2. Suponga que tiene implementado en RCFAE las operaciones de listas de scheme: `'()`, `car`, `cdr`, `cons` con la sintáxis `-empty`, `-car`, `-cdr`, `-cons` respectivamente. Dado el siguiente programa:

```
(interp (parse
          '{rec {ones {-cons 1 ones}}
             {rec {take {fun {n}
                  {if0 n {-empty}
                       {-cons {-car ones} {take {- n 1}}}}}}}
           {take 8}}}) (mtSub))
```

   - ¿Cuál es el resultado del programa si RCFAE es eager?
     Respuesta: El programa cae en un loop infinito si es eager porque en la definición de `ones` se usa la función `-cons` que recibe de argumento `ones` y no hay condición de término. Como en eager los argumento son evaluados antes de la aplicación, ones es evaluado infinitamente.
   - ¿Cuál es el resultado del programa si RCFAE es lazy?
     Respuesta: El resultado es una lista de 8 unos (`(1,1,1,1,1,1,1,1)`). Al ser lazy, los argumentos de `-cons` en la definición de `ones` serán sólo evaluados cuando es necesario, en particular cuando se necesita el primer elemento en el ejemplo (uso de `-car` en la definición de `take`)

   En ambos casos justifique su respuesta.