Lenguajes de Programación
Nota de clase 4: Funciones y Alcance

Karla Ramírez Pulido Manuel Soto Romero

26 de octubre de 2021
Facultad de Ciencias UNAM

El Cálculo λ estudiado en la Nota de Clase 3, ha servido como base de muchos lenguajes de programación funcionales como LISP y HASKELL; el principal aspecto que se hereda del Cálculo λ es el uso de funciones como miembros de primera clase. En esta nota se estudia el concepto de función en lenguajes de programación, una taxonomía para su clasificación, así como las diferencias entre alcance estático y dinámico.

Para estudiar estos conceptos se extiende el lenguaje WAE para que soporte funciones, la gramática de este nuevo lenguaje FWAE (*Functions, With and Arithmetic Expressions*), es la siguiente:

```
<expr> ::= <id>
         | <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}
```

De esta forma es posible definir funciones, por ejemplo, la función identidad:

```
{fun {x} x}
```

Así como en el Cálculo λ, estas funciones son anónimas. La primitiva **fun**, representa una abstracción λ, en este caso a la abstracción $(\lambda x.x)$. De la misma forma, son equivalentes a las funciones **lambda** del lenguaje RACKET (**λ (x) x**). También es posible aplicarlas con un argumento, por ejemplo, la función identidad con el argumento **2**:

```
{{fun {x} x} 2}
```

Al igual que en el Cálculo λ, en una expresión de la forma **{e1 e2}**, se dice que **e1** está en la primera posición correspondiente a la definición de una función y que **e2** está en la segunda posición la cual corresponde argumento que tomará la función (parámetro real).

4.1. Taxonomía de Funciones

Las funciones de FWAE se definen de la misma forma que en el Cálculo λ, esto es, una función puede crear otras funciones, pueden usarse como cualquier otro valor en el lenguaje; sin embargo, no todos los lenguajes de programación ofrecen estas capacidades (JAVA, C, FORTRAN, ALGOL, etc.). De esta forma Shriram Krishnamurthi, propone, una taxonomía que permite clasificar a las funciones de acuerdo a su comportamiento en un lenguaje [2]:

Funciones de Primer Orden. Este tipo de funciones no son consideradas como valores en los lenguajes de programación. Son definidas para ser usadas en determinadas secciones de un programa y son llamadas a partir de un nombre dado para éstas.

Funciones de Orden Superior En los lenguajes con este tipo de funciones, es posible que una función devuelva una nueva función como resultado.

Funciones de Primera Clase Las funciones son consideradas como valores, en particular, éstas pueden pasarse como parámetro a otras funciones, regresar nuevas funciones como resultado y almacenarse en estructuras de datos.

4.2. Interpretación de Funciones

Para definir funciones, puede usarse el algoritmo de sustitución definido previamente. La función **interp** correspondiente se muestra en el Código 1.

```
1  ;; interp: FWAE → FWAE
2  (define (interp expr)
3     (type-case FWAE expr
4        [id (i)
5           (error 'interp "Variable libre")]
6        [num (n)
7           expr]
8        [(add lhs rhs)
9           (num (+ (num-n (interp lhs)) (num-n (interp rhs))))]
10       [(sub lhs rhs)
11          (num (- (num-n (interp lhs)) (num-n (interp rhs))))]
12       [(with id value body)
13          (interp (subst body id value))]
14       [(fun param body)
15          expr]
16       [(app fun-expr arg)
17          (interp (subst (fun-body fun-expr)(fun-param fun-expr) (interp arg)))]))
```

Listado de código 1: Evaluación de funciones mediante sustitución

En el Código 1, se puede apreciar que la firma de la función **interp** cambia con respecto a la última implementación establecida en el lenguaje WAE (Nota de Clase 2). Esta nueva versión regresa expresiones del propio FWAE debido a que al manejar funciones de primera clase, éstas pueden devolverse como resultado de su evaluación, con lo cual se debe unificar el tipo de regreso de la función para soportar números y funciones y el tipo más adecuado para esto, resulta ser FWAE.

De esta forma la interpretación de números (líneas 6 y 7) y de funciones (líneas 14 y 15) resulta ser similar, ya que devuelve la expresión recibida. Por ejemplo, al introducir la siguiente expresión en el intérprete[1]:

[1] En realidad, esta expresión es procesada por el analizador sintáctico antes de ser interpretada.

```
{with {foo {fun {x} x}}
   foo}
```

Se obtiene:

```
(fun 'x (id 'x))
```

Por otro lado, la aplicación de funciones (líneas 16 a 20) se interpreta de forma similar al proceso de β-reducción del Cálculo λ, debe sustituirse el parámetro de la función por el argumento en el cuerpo de la misma y se interpreta nuevamente hasta llegar al resultado final (Forma Normal en el Cálculo λ).

La modificación de la función **subst** para procesar funciones y su aplicación se deja como ejercicio al lector.

4.3. Azúcar sintáctica

Los casos de **interp** del Código 1 para la evaluación de expresiones **with** (líneas 12 y 13) y de la aplicación de funciones (líneas 16 a 20), presentan un comportamiento similar. Por ejemplo, la siguiente expresión **with**:

```
{with {a 2}
   {+ a 3}}
```

Se evalúa a **(num 5)** al sustituir la variable por su valor en el cuerpo del **with**. Por otro lado, la expresión:

```
{{fun {a} {+ a 3}} 2}
```

Se evalúa también a **(num 5)** al sustituir el parámetro de la función por el argumento recibido en el cuerpo de la misma. De esta forma, se puede apreciar que la primitiva **with** es un caso particular de la aplicación de función y por lo tanto son semánticamente equivalentes. En este caso se dice que **with** es azúcar sintáctica de la aplicación de función.

Con esto, es posible eliminar la primitiva **with** y simplemente realizar la conversión correspondiente:

```
{with {<id> <value>} <body>} ⇒ {{fun {<id>} <body>} <value>}
```

Lo cual simplifica los casos de la función **interp**. El Código 2, muestra esta nueva versión del intérprete sin el caso de **with**, mediante el tipo de dato FAE.

Ejercicio 1. Se deja como ejercicio al lector la implementación de la función **desugar: FWAE → FAE** que elimina el azúcar sintáctica de las expresiones **with** y las mapea al nuevo tipo de dato **FAE**.

```
1 ;; interp: FAE → FAE
2 (define (interp expr)
3    (type-case FWAE expr
4       [id (i)
5          (error 'interp "Variable libre")]
6       [num (n)
```

```
7           expr]
8        [(add lhs rhs)
9           (num (+ (num-n (interp lhs)) (num-n (interp rhs))))]
10       [(sub lhs rhs)
11          (num (- (num-n (interp lhs)) (num-n (interp rhs))))]
12       [(fun param body)
13          expr]
14       [(app fun-expr arg)
15          (interp (subst (fun-body fun-expr)(fun-param fun-expr) (interp arg)))]))
```

Listado de código 2: Eliminación de with

4.4. Ambientes de evaluación

Dada la siguiente expresión:

```
{with {x 3}
   {with {y 4}
      {with {z 5}
         {+ x y}}}}
```

Para ser evaluada es necesario llamar varias veces al algoritmo de sustitución. La primera sustitución `[x := 3]` se aplica 3 veces, pues tiene dos expresiones **`with`** anidadas y el cuerpo del **`with`** más interno. La segunda sustitución `[y := 4]` se aplica 2 veces y la sustitución `[z := 5]` una vez.

De forma general, si el programa tuviera n variables, se tendrían:

$$n + (n-1) + (n-2) + ... + 3 + 2 + 1 = \frac{n\,(n-1)}{2}$$

sustituciones. Con lo cual la complejidad del intérprete definido es $O\,(n^2)$.

Se necesita entonces una forma de almacenamiento para dichas sustituciones con el objetivo de poder usarlas conforme se vayan solicitando en el programa. Cada que se encuentre una expresión **`with`** o una aplicación de función, debe guardarse la sustitución correspondiente y buscarla cuando se solicita el valor de una variable.

Una posible implementación puede ser usando una pila (*stack*), donde la búsqueda de variables se da de forma lineal ($O\,(n)$). A esta pila se le conoce como *ambiente de evaluación.* Por ejemplo, el ambiente para la expresión anterior:

Figura 1: *Un Ambiente de Evaluación*

Cada elemento en el ambiente se compone de una asignación (identificador, valor), la cual está representada mediante el identificador de la variable, y el valor asociado a la misma. El valor puede ingresar al ambiente ya evaluado cuando se usa un régimen de evaluación glotón o sin evaluar (en forma de expresión) cuando se usa un régimen de evaluación perezoso. Por ejemplo, dada la siguiente expresión:

```
{with {x 3}
   {with {y {+ x x}}
      {+ x y}}}
```

El ambiente de evaluación bajo un régimen de evaluación glotón y perezoso se puede observar en la Figura 2.

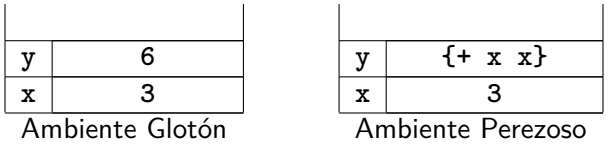

Figura 2: *Ambientes de evaluación en distintos régimenes*

Para implementar ambientes, se muestra el Código 3, donde la definición del tipo de dato ambiente **Env** es:

```
1 (define-type Env
2    [mtSub]
3    [aSub (id symbol?) (value FAE?) (rest-env Env?)])
```

Listado de código 3: Ambientes de evaluación

El primer constructor (línea 2) representa los ambientes de evaluación sin sustituciones[2], es decir, el ambiente vacío o inicial, mientras que el segundo constructor (línea 3) representa una sustitución[3] con su identificador y valor asociado, junto con el resto del ambiente pues se definen de forma recursiva.

La búsqueda de valores en el ambiente se da mediante la función **lookup** definida en el Código 4.

```
1 ;; lookup: symbol Env → FAE
2 (define (lookup sub-id env)
3    (type-case Env env
4       [mtSub ()
5          (error 'lookup "Variable libre")]
6       [aSub (id value rest-env)
7          (if (symbol=? id sub-id)
8              value
9              (lookup sub-id rest-env))]))
```

Listado de código 4: Búsqueda de variables

[2] **mtSub**, del inglés *Empty Substitution*.
[3] **aSub**, del inglés *A Substitution*.

Si una sustitución no es encontrada en el ambiente de evaluación (líneas 4 y 5), la función **lookup** devuelve un error ya que se trata de una variable libre. Si por el contrario el ambiente no está vacío (líneas 6 a 9), se compara el identificador que buscamos con cada identificador almacenado en el ambiente y si es el mismo identificador (símbolo), entonces se regresa su valor asociado; en caso contrario, se continúa con la búsqueda recursivamente.

De esta forma, la función **interp** se modifica usando ambientes de evaluación, como se muestra en el Código 5.

```
1  ;; interp: FAE Env → FAE
2  (define (interp expr env)
3     (type-case FWAE expr
4        [id (i)
5           (lookup i env)]
6        [num (n)
7           expr]
8        [(add lhs rhs)
9           (num (+ (num-n (interp lhs env)) (num-n (interp rhs env))))]
10       [(sub lhs rhs)
11          (num (- (num-n (interp lhs env)) (num-n (interp rhs env))))]
12       [(fun param body)
13          expr]
14       [(app fun-expr arg)
15          (let ([fun-val (interp fun-expr)])
16             (interp (fun-body fun-val)
17                     (aSub (fun-param fun-val)
18                           (interp arg)
19                           env)))]))
```

Listado de código 5: Evaluación mediante ambientes de evaluación

Se modifica la firma de la función a **interp: FAE Env → FAE** de tal manera que se tenga el ambiente como parámetro en todo momento.

La búsqueda de variables se da para el caso de un identificador (líneas 4 y 5), de esta forma la función **lookup** es la encargada de reportar cuando se tiene una variable libre. Por otro lado, la interpretación de aplicaciones de función (líneas 14 a 19) consiste en evaluar el cuerpo de la función correspondiente en el ambiente formado por el ambiente actual después de añadir la sustitución (asignación) formada por el parámetro de la función y su argumento.

4.5. Alcance Estático y Dinámico

Dada la siguiente expresión:

```
{with {x 3}
   {with {f {fun {y} {+ x y}}}
      {with {x 5}
         {f 4}}}}
```

Al ser evaluada por el intérprete basado en sustitución (Código 2) se obtiene un resultado de 7, mientras que con el intérprete basado en ambientes (Código 5) se obtiene un resultado de 9. Esto ocurre debido al tipo de alcance que usan cada uno de los intérpretes. A continuación se presentan dos definiciones [2]:

Definición 1. (Alcance Estático) *En un lenguaje con alcance estático, el alcance de un identificador es la región en la cual se encuentra definido el mismo.*

Por ejemplo, el intérprete basado en sustitución usa alcance estático pues en la expresión anterior, el valor de `x` dentro de la función `{fun {y} {+ x y}}` se toma de la región que delimita a esa función, es decir, 3.

Definición 2. (Alcance Dinámico) *En un lenguaje con alcance dinámico, el alcance de un identificador es todo el programa, es decir, se toma la última asignación hecha al mismo.*

Por ejemplo, un intérprete basado en ambientes usa alcance dinámico pues en la expresión anterior, el valor de `x` dentro de la función `{fun {y} {+ x y}}` se toma de la ultima asignación hecha, es decir 5.

A continuación se analiza otro ejemplo:

```
{with {g {fun {x} {+ x 2}}}
   {with {f {fun {y} {g y}}
      {with {g {fun {x} {+ x 4}}}
         {f 7}}}}}
```

Para analizar el alcance que toma esta expresión, se usarán ambientes de evaluación, la búsqueda se realiza usando el concepto del alcance estático:

1. Ambiente inicial

Se construye el ambiente con las sustituciones (asignaciones) encontradas.

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

2. Se evalúa el cuerpo del with más anidado, es decir, {f 7}

El cuerpo del `with` *es una aplicación* `{f 7}`*, por lo tanto se busca desde la posición actual (el tope del ambiente) el valor de la función* `f`*.*

$\Rightarrow$

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

3. Ingresa una nueva sustitución (asignación)

Dado que el parámetro de `f` *es* `y` *y su parámetro real es 7, entonces se añade la sustitución* `[y := 7]` *al ambiente, a partir de la definición de la función.*

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
y	7
g	{fun {x} {+ x 2}}

⇒

4. Búsqueda de g

El cuerpo de la función `f` *evalúa la función* `g` *con su parámetro 7. Posicionados en la definición de* `f`*, inicia la búsqueda de* `g`*.*

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
y	7
g	{fun {x} {+ x 2}}

⇒

5. Ingresa una nueva sustitución (asignación)

Dado que el parámetro de `g` *es* `x` *y que ésta se mandó llamar con 7, se agrega la sustitución de* `[x := 7]` *al ambiente a partir de la definición de la función.*

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
y	7
g	{fun {x} {+ x 2}}
x	7

⇒

6. Resultado

El cuerpo de la función `g` *evalúa la expresión* `{+ 7 2}` *la cual da por resultado el valor de* `9`*.*

Por otro lado, al usar alcance dinámico. se tiene lo siguiente:

1. Ambiente inicial

Se construye el ambiente con las sustituciones (asignaciones) encontradas.

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

2. Se evalúa el cuerpo del with más anidado, es decir, {f 7}

El cuerpo del `with` *es una aplicación* `{f 7}`*, por lo tanto se busca desde la posición actual (el tope del ambiente) el valor de la función* `f`*. se encuentra y se obtiene el valor de éste, que en este caso es la función* `{fun {y} {g y}}`*.*

$\Rightarrow$

g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

3. Se añade una nueva sustitución (asignación) al ambiente

Dado que el parámetro de `f` *es* `y` *y que ésta se mandó llamar con el valor de 7, se agrega la sustitución o asignación* `[y := 7]`.

$\Rightarrow$

y	7
g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

4. Se hace la búsqueda del identificador g

En el cuerpo de la función `f` *se requiere la evaluación de la aplicación* `{g y}`. *Por lo que tanto el identificador* `g` *debe estar en el ambiente y por otro lado* `y` *también. En el caso de* `g` *se busca su valor (que es una función) y de* `y` *se obtiene el valor de 7.*

$\Rightarrow$

y	7
g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

5. Se añade una nueva sustitución

Dado que el parámetro de `g` *es* `x` *y que ésta se mandó tiene como parámetro real 7, se agrega la sustitución* `[x := 7]` *al ambiente.*

$\Rightarrow$

x	7
y	7
g	{fun {x} {+ x 4}}
f	{fun {y} {g y}}
g	{fun {x} {+ x 2}}

6. Resultado

El cuerpo de la función `g` *evalúa* `{+ 7 4}` *el cual da por resultado el valor numérico de 11.*

4.5.1. Implementación

Para modificar el intérprete basado en ambientes de tal manera que implemente alcance estático, es necesario que las funciones usen el ambiente cuando éstas fueron definidas, de forma que no utilicen el ambiente actual completo pues esto haría que la búsqueda comenzara desde el tope de éste.

Una forma de representar funciones es través de *cerraduras de función* o simplemente *cerraduras*[4].

Una cerradura es una estructura, la cual almacena el parámetro, el cuerpo de la función y el ambiente donde ésta fue definida, este último puede ser visto como un subambiente [2]. De esta forma, al evaluar una función en lugar de simplemente regresarla, se debe regresar una cerradura que *encierra* a la función con el ambiente actual. Por lo que se debe cambiar el tipo de regreso de la función **`interp`** para que maneje valores como (1) cerraduras y (2) números.

El Código 6 muestra la definición del tipo de dato **`FAE-Value`**. el cual se usa para definir constructores de valor, número (**`numV`**) y cerraduras (**`closureV`**).

```
1 (define-type FAE-Value
2    [numV     (n number?)]
3    [closureV (param symbol?) (body FAE?) (env Env?)])
```

Listado de código 6: Tipo de regreso de interp

La función **`interp`** se define en en el Código 7, el cuál implementa alcance estático.

```
1  ;; interp: FAE Env → FAE-Value
2  (define (interp expr env)
3     (type-case FWAE expr
4        [id (i)
5           (lookup i env)]
6        [num (n)
7           (numV n)]
8        [(add lhs rhs)
9           (numV (+ (numV-n (interp lhs env)) (numV-n (interp rhs env))))]
10       [(sub lhs rhs)
11          (numV (- (numV-n (interp lhs env)) (numV-n (interp rhs env))))]
12       [(fun param body)
13          (closureV param body env)]
14       [(app fun-expr arg)
15          (let ([fun-val (interp fun-expr)])
16             (interp (closureV-body fun-val)
17                     (aSub (closureV-param fun-val)
18                           (interp arg)
19                           (closureV-env fun-val)))])))
```

Listado de código 7: Evaluación con alcance estático

La interpretación para números (líneas 6 y 7) cambia con el fin de usar el nuevo tipo de dato **`numV`**. La interpretación de funciones (líneas 12 y 13) construye una cerradura, con los datos de la función y el ambiente actual. Finalmente

[4]Traducción del término en inglés *closure*.

también se modifica el caso de la aplicación de función (líneas 15 a 19) de manera que se evalúe el cuerpo de la función construido utilizando así, el ambiente formado por la sustitución del parámetro dentro de la cerradura y su argumento usando ahora el ambiente de la cerradura en lugar del actual. Cuando se busca un identificador en el ambiente donde fue definida la función éste deberá encontrarse (en caso de haberse definido previamente) en el ambiente contenido dentro de la cerradura.

4.6. Ejercicios

4.6.1. Sintaxis de Funciones

1. Dada la sintaxis concreta y abstracta del lenguaje FWAE:

   - Muestra una derivación de las siguientes expresiones o justifica por qué éstas no son parte de la gramática.
   - En caso de ser posible obtén su correspondiente sintaxis abstracta.

   (a) `{fun {x} {+ x x}}`
   (b) `{fun {} {+ 2 3}}`
   (c) `{fun {x y} {+ x y}}`
   (d) `{{fun {x} {+ x x}} 3}`
   (e) `{{fun {x} x} 2 3}`

2. Elimina el azúcar sintáctica de las siguientes expresiones:

   (a) `{with {a 2} {+ a a}}`
   (b) `{with {b 3} {with {x 8} {+ b x}}}`

4.6.2. Alcance Estático y Dinámico

1. Evalúa las siguientes expresiones usando (a) alcance estático y (b) alcance dinámico. Es necesario mostrar el ambiente de evaluación usando representación de pila.

   (a)
```
{with {x -2}
      {with {x 4}
         {with {g {fun {y} {+ x y}}}
            {with {f {fun {z} {- z w}}}
               {with {x 3}
                  {with {x 2}
                     {f 10}}}}}}}
```

   (b)
```
{with {a 1}
      {with {b {+ 1 a}}
         {with {c {+ b a}}
            {with {foo {fun {x} {+ c {+ b x}}}}
               {with {super-foo {fun {x} {+ a {- x a}}}}
                  {with {a 10}
                     {with {turbo-foo {fun {x} {+ 1 {+ a x}}}}
                        {with {a -1}
                           {turbo-foo 1}}}}}}}}}
```

(c)
```
{with {a -1}
      {with {b 1}
         {with {a 1}
            {with {foo1 {fun {x} {+ a b}}}
               {with {b 0}
                  {with {a 1}
                     {with {foo2 {fun {x} {- {+ a b} b}
                        {with {a -2}
                           {with {b 3}
                              {foo1 4}}}}}}}}}}}
```

Referencias

[1] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera Edición, 2007.

[2] Daniel P. Friedman, Mitchell Wand *Essentials of Programming Languages* Third Edition, The MIT Press, 2008.

[3] Éric Tanter, *A Note on Dynamic Scope.* Versión 02-04-2016.