**TEMA 2: TECNICAS DE PLANIFICACION Y BUSQUEDA**

En este tema vamos a ver algunas de las técnicas de planificación partiendo de la programación PDDL que se ha visto en el tema anterior.

# Contenidos

1. SSS & PSP

1.1. Introducción

1.2. Clasificación de algoritmos: búsqueda en espacio de estados(SSS)

1.2.1 SSS mediante progresión

1.2.2 SSS mediante regresión

1.3. Clasificación de algoritmos: búsqueda en espacio de planes(PSP)

1.4. Técnicas de planificación:

2. Planificadores de Orden Total(TOP)

2.1. Definición

2.2. Plan prodigios

3. Planificadores de Orden Parcial(POP)

3.1. Definición

3.2. Árbol POP

3.3. UCPOP

3.4. Planificadores POP

4. Funciones de orden superior

4.1. FOS en la clase List

4.2. Funciones genéricas

4.3 Método foldRight

5. Ámbitos

5.1. Reglas de evaluación de expresiones Scala con ámbitos

5.2. Clausuras en Scala

6. Otras características funcionales de Scala

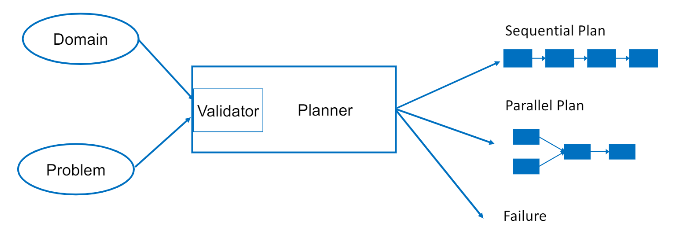
6.1. Definición por comprensión de colecciones

6.2. *Currying*

# 1. SSS & PSP

## 1.1. Introducción

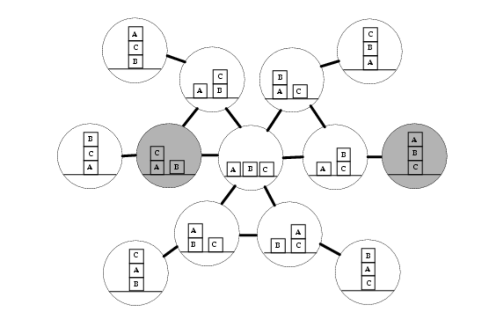
Casi todos los procedimientos de planificación son procedimientos de búsqueda. Como ya vimos en el tema anterior, cuando deseamos resolver un problema de planificación, disponemos de un fichero dominio.pddl y de un fichero problema.pddl. Estos ficheros antes d ser ejecutados, pasan por un validador que comprueba que no haya ningún tipo de error en el código. Una vez que sabemos que no hay errores, el planificador se encarga de generar una solución. Dicha solución o plan, puede ser de tres tipos, secuencial, paralela, y error.



Cuando el planificador nos proporciona un plan secuencial, las tareas se realizan una detrás de otra, mientras que, si nos proporciona un plan paralelo, varias tareas pueden hacerse al mismo tiempo. Por último, el planificador puede que no encuentre una solución o bien porque nuestro problema esta mal descrito o bien porque el planificador no soporta la versión de pddl que estamos empleando.

## 1.2. Clasificación de algoritmos: búsqueda en espacio de estados

En la búsqueda en espacio de estados, cada nodo representa un estado del mundo y el plan es una camino a través del dicho espacio. En el plan de estados, cada nodo es un conjunto de operadores parcialmente instanciados con algunas restricciones. Se trata de imponer cada vez mas y mas restricciones hasta encontrar un plan.

La madera más sencilla de construir un planificador es convertirlo en un problema de búsqueda a través de un espacio de estados. Donde como ya se ha indicado antes, cada nodo en el plan representa un estado del mundo y los arcos que conectan cada nodo son acciones. Una vez que el problema se ha convertido, podemos aplicar cualquier algoritmo estudiado.

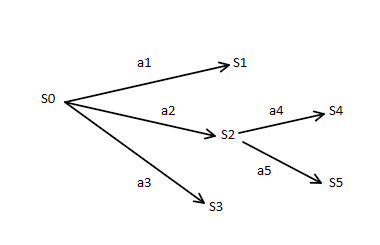
De manera incremental, podemos ir generando un conjunto de estados accesible desde el estado inicial, mediante una secuencia de acciones, mediante **progresión**. Los estados alcanzados en la secuencia pueden ser calculados comprobando si la meta ha sido alcanzada o si las precondiciones se han satisfecho.

Pero en vez de mirar hacia delante desde el estado inicial, podemos volver hacia atrás desde las metas encontrando así una solución, mediante **regresion**.

Una vez que disponemos de nuestro algoritmo, debemos tener en cuenta varias consideraciones. En primer lugar, debemos saber si dicho algoritmo es robusto, es decir retorna una solución valida. En segundo lugar, si es completo, es decir si la meta que se propone puede ser alcanzada.

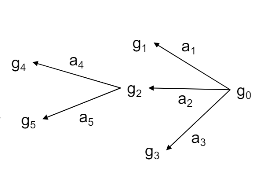
## 1.2.1. Búsqueda en espacio de estados mediante progresión

Para cualquier plan devuelto por alguna ruta no determinista, este plan está garantizado que será una solución. La búsqueda mediante progresión será completa si existe un camino que llegue a la solución desde el estado inicial. Algunas implementaciones de la búsqueda en espacio de estados mediante progresión son: BEA, BEEP, A\*…

BEA y A\* son algoritmos óptimos y completos, pero no se suelen utilizar mucho en la practica puesto que requieren mucha memoria. La memoria requerida es exponencial al tamaño de la solución. En la practica el algoritmo mas utilizado es BEEP debido a que la memoria requerida es lineal al tamaño de la solución, en general es óptimo, pero no completo.

La búsqueda en espacio de estados mediante progresión puede tener un factor alto de ramificación, esto es peligroso puesto que implementaciones deterministas pueden malgastar mucho tiempo en acciones irrelevantes. Es necesario una buena función heurística y/o un proceso de poda.

## 1.2.1. Búsqueda en espacio de estados mediante regresion

En la búsqueda mediante progresión, comenzamos desde el estado inicial y media te una serie de estados tratamos de llegar a la meta. En la búsqueda mediante regresion, comenzamos desde la meta y vamos hacia atrás en busca del estado inicial.

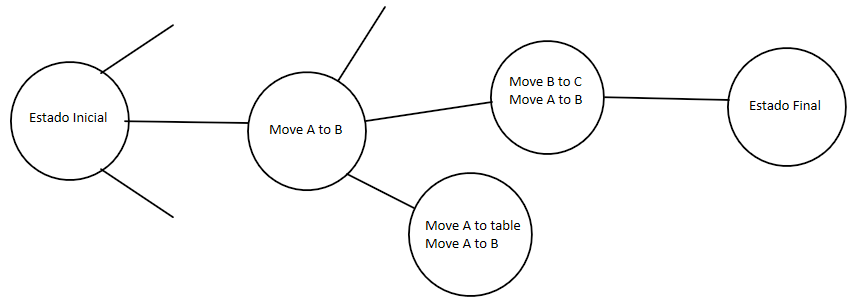
La búsqueda por regresion también puede tener un factor alto de ramificación, al igual que antes, implementaciones deterministas pueden malgastar mucho tiempo. STRIPS usa esta búsqueda.

## Lifting

Podemos reducir ese factor de ramificación de la búsqueda hacia detrás si inicialmente instanciamos los operadores. El factor de ramificación seguirá siendo alto, pero mucho menos que antes. Puede que existe un plan que no tenga solución, en el que no podamos llegar al estado inicial.

## 1.3. Clasificación de algoritmos: búsqueda en espacio de planes

En 1974 se construyó el planificador NOAH, la búsqueda conducía a través de un espacio de planes. En dicha búsqueda cada nodo en el plan/árbol representa un plan parcial y cada arista representa operaciones de dicho plan. Además, en el plan puede haber un conjunto de restricciones.

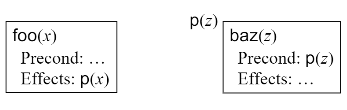
El nodo inicial del plan es NULL y el nodo final representa el plan solución, la meta. Mientras que la búsqueda en espacio de estados retorna el camino desde el estado inicial hasta la meta, en la búsqueda en espacio de planes, la meta es la solución.

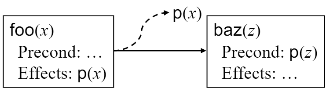
En cuanto a las restricciones, existen distintos tipos que son los siguientes:

Restricciones de precedencia: A debe preceder a B.

Restricciones de unión: pueden ser restricciones de desigualdad(≠) o de igualdad(=)

Enlaces causales: usar las acción A para establecer las precondiciones que necesita la acción B.

Existen ciertos problemas, algunos de ellos son los **flaws** y los **threat**. Los flaws son casos donde parecer metas abiertas, es decir, acciones que tienes precondiciones que no han sido establecidas. Uno de esos caso son lo que se denomina como flaws. Una manera de resolver este problema es usar una determinada acción *b* que podemos insertar nosotros en el plan, para establecer una acción *p*.



Un threat o amenaza, es una interacción para eliminar una condición. Por ejemplo, una determinada acción *a* establece una precondición de la acción *b*. Otra acción *c* es capaz de eliminar *p*. Para resolver este problema, creamos una restricción que impida a *c* eliminar *p*, ¿Cómo?, hacemos que *b* preceda a *c*, y que *c* preceda a *a*.

## 

## 1.3. Técnicas de planificación

A continuación, y con más profundidad en los siguientes apartados, se explicarán una serie de técnicas de planificación. Alguna de ellas son las siguientes:

TO: La solución es una secuencia totalmente ordenada de acciones(estados/planes).

POP: Búsqueda en espacio de planes. Implementa lo que se denomina como “leats Commited approach”.

HTN: redes de tareas y restricciones.

Grafo-base(GP): la estructura de búsqueda es un Grafo de Planificación.

SAT: toma un problema como entrada, determina su tamaño y genera una serie de cláusulas preposicionales.

HSP: transforma problemas de planificación en problemas de heurística.

# 2. Planificadores de Orden Total

## 2.1. Definición

Como ya hemos dicho anteriormente, la solución de todo planificador es una secuencias de acciones que están ordenadas. Un plan es válido si las precondiciones de cada operador se satisfacen antes de la ejecución de dicho operador. A diferencia de algoritmos que veremos mas adelantes, introducir restricciones so es necesario. Dado un determinado problema, la mayoría de planificadores empiezan con un plan vacío y lo van modificando a medida que se encuentra la solución.

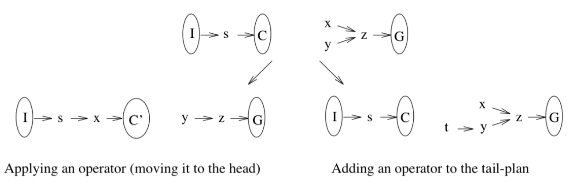
Un plan puede ser modificado insertando nuevos operadores, introduciendo restricciones o instanciando variables en la descripción de un determinado operador.

Durante la ejecución del plan, mientras que se determina la solución, el plan se considera incompleto. Cada plan incompleto puede ser visto como un nodo en la búsqueda del espacio de planes del algoritmo. Modificar un plan incompleto consiste con expandir un nodo.

El factor de ramificación se determina por el numero de posibles modificaciones del plan.

La distinción de planificadores de orden total debe estar separada de la distinción de los planificadores SSS y PSP. Como ya hemos vito en el apartado anterior, en SSS cada estado de la búsqueda corresponde con un estado en el mundo, mientras que en PSP cada estado de la búsqueda corresponde con un plan. Generalmente TO se corresponde con SSS.

## 2.2. Prodigy planner

Un plan incompleto consiste de dos partes, una cabeza y una cola. La cola del plan se construye mediante un algoritmo de regresion, que comienza en la meta y añade operadores uno por uno para conseguir que dichos operadores satisfagan sus precondiciones, los operadores que no se satisfacen sus precondiciones se denominan metas pendientes .La cabeza del plan aplica el operador en la cola del plan.

# 3. Planificadores de Orden Parcial

## 3.1. Definición

Los planificadores de orden parcial se basan en la planificación en espacio de planes. Un plan parcial se define como P = (A, O, L, OC, UL), es decir, está compuesto por acciones, restricciones ordenadas, enlaces causales, condiciones abiertas(flaws), enlaces inseguros.

Los planificadores de orden parcial realizan la búsqueda mediante regresion, donde cada nodo está compuesto por acciones parcialmente instanciadas y un conjunto de restricciones. El proceso para si la solución se encuentra.

El algoritmo de planificación implementa leats commited technique. Solo las acciones esenciales son guardadas puesto que aquellas que no son necesarias no se comitean. La estructura del enlace causal es la responsable de almacenarlas en tres campos, productos, consumidor y proposición.

Algunos ejemplos de planificadores de orden parcial son UCPOP, Cassandra, ZENO, VHPOP.

## 3.2. Árbol POP

El plan inicial es creado por la descripción del estado inicial y la descripción de la meta. Dicho plan inicial se crea en dos pseudo pasos:

Comienzo:

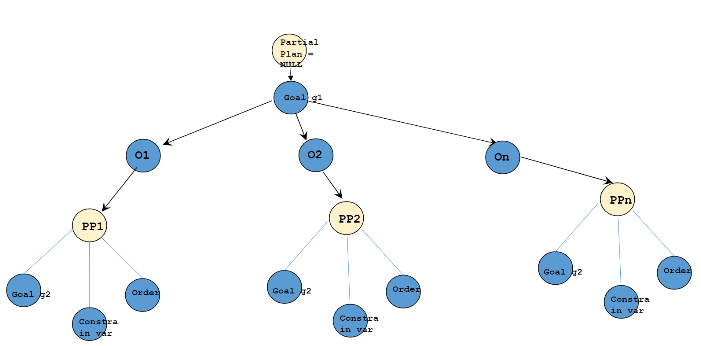
Precondición: Ninguna

Efecto: Todas los literales positivos se definen en el estado inicial

Final:

Precondición: Los literales definen conjuntamente la meta a alcanzar

Efecto: Ninguno



## 3.3. Planificadores de Orden Parcial

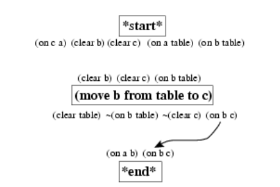
UCPOP

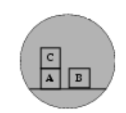
Cassandra

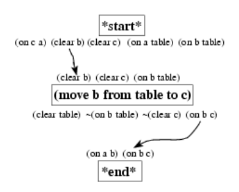
ZENO

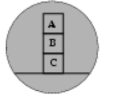
VHPOP

## 3.4. Ejemplo: UCPOP

Partimos del ejemplo de blocksworld donde disponemos de tres cajas iniciales y tenemos que disponerlas de una determina forma.



Como podemos observar, se ha añadido un enlace causal entre la acción “mover b de la mesa a c” y la meta. Después de ello, unimos podemos unir el estado inicial con dicha acción y encontrar un camino.



Una vez añadido el ultimo enlace causal de la acción (clear b), el plan contiene dos enlaces causales {(clear c) (on B table) (on A B)}

# 4. Funciones de orden superior

La definición de funciones como un tipo de dato primitivo permite definir *funciones de orden superior* que toman como parámetro otras funciones y permiten hacer mucho más conciso y general el código escrito en Scala.

## 4.1. FOS en la clase List

Por ejemplo, en la clase List se definen métodos como los siguientes, que admiten funciones como parámetro.

count : cuenta el número de elementos de la lista que satisfacen un predicado exists : comprueba si algún elemento de la lista satisface un predicado filter : devuelve una nueva lista con los elementos de la lista original que cumplen el

predicado map : aplica una función a todos los elementos de la lista, devolviendo una nueva lista con el

resultado foldRight : pliega la lista con una función de plegado, devolviendo un único dato como

resultado

Son métodos definidos como genéricos que se pueden aplicar a listas de distintos tipos. El tipo de la función que se pasa como argumento tiene que corresponder con el tipo de los elementos de la lista.

Veamos distintas formas de llamar a estas funciones genéricas, utilizando

count

como ejemplo:

val lista = List(1,2,3,4,5,6,7,8,9)

lista.count((x:Int) => {x % 2 == 0})

⇒

Int = 4

Podemos escribir la función que pasamos como parámetro de

count

de distintas formas, cada

una de ellas más concisa. El compilador de Scala se encarga de completar lo necesario:

lista.count((x) => {x % 2 == 0})

lista.count(x => x % 2 == 0)

lista.count(\_ % 2 == 0)

podemos definir la función

par

y pasarla como parámetro:

def par(x:Int): Boolean = {x % 2 == 0}

lista.count(par \_)

Algunos ejemplos más:

lista.filter(par \_)

⇒

List[Int] = List(2, 4, 6, 8)

def cuadrado(x:Int) = x\*x

lista.map(cuadrado \_)

⇒

List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81)

List("No","es","elegante","escribir","con","mayusculas").map(s => s.toUpperC

ase)

⇒

List[String] = List(NO, ES, ELEGANTE, ESCRIBIR, CON, MAYUSCULAS)

Al igual que hacíamos en Scheme, es importante no sólo saber utilizar estas funciones sino también su implementación recursiva.

Función miCount , que cuenta el número de elementos de una lista de tipo Int que cumplen un predicado:

def miCount(lista:List[Int], p:(Int)=>Boolean): Int = {

if (lista == Nil) 0 else

if (p(lista.head)) 1+miCount(lista.tail,p) else

miCount(lista.tail,p)

}

Función

miMap

, que aplica una función a todos los elementos de una lista:

def miMap(lista:List[Int], f:(Int)=>Int): List[Int] = {

if (lista == Nil) Nil else

f(lista.head) :: miMap(lista.tail,f)

}

## 4.2. Funciones genéricas

En Scala también es posible definir funciones con tipos de datos genéricos. Por ejemplo, la siguiente función miFilter utiliza el tipo genérico A como el tipo de los elementos de la lista y el tipo del predicado que se aplica a los elementos.

El compilador determina el tipo

A

cuando se invoca a la función con una lista y un predicado

concreto.

def miFilter[A](lista: List[A], pred: (A)=>Boolean): List[A] = {

if (lista.isEmpty) Nil else

if (pred (lista.head)) lista.head :: miFilter(lista.tail, pred)

else miFilter(lista.tail, pred)

}

def par(x: Int) = x % 2 == 0

miFilter(List(1,2,3,4), par \_)

⇒

List[Int] = List(2, 4)

miFilter(List("hola","amigo","adios"),(s: String) => { s.head == 'a'})

⇒

List[String] = List(amigo, adios)

Las versiones genéricas de las funciones recursivas sobre listas que vimos anteriormente son:

def numElems[A](lista:List[A]):Int =

if (lista.isEmpty) 0 else

1 + numElems(lista.tail)

def reverse[A](lista: List[A]) : List[A] =

if (lista.isEmpty) Nil else

reverse(lista.tail) ::: List(lista.head)

def insert[A](x: A, lista: List[A], menor: (A,A) => Boolean) : List[A] =

if (lista == Nil) x :: Nil else

if (menor(x,lista.head)) x :: lista else

lista.head :: insert(x, lista.tail, menor)

def sort[A](lista: List[A], menor: (A,A) => Boolean): List[A] =

if (lista.isEmpty) Nil else

insert(lista.head, sort(lista.tail, menor), menor)

Por ejemplo, podemos definir una ordenación entre tuplas de dos enteros basada en comparar la suma de sus dos componentes. Y después, utilizarla para ordenar una lista de tuplas:

def menorTuplas(t1: (Int,Int), t2: (Int,Int)) = t1.\_1+t1.\_2 < t2.\_1+t2.\_2

sort(List((1,2),(3,1),(-1,-2)), menorTuplas)

⇒

List[(Int, Int)] = List((-1,-2), (1,2), (3,1))

**4.3**

**Método**

**foldRight**

Hay que hacer una mención especial al método de orden superior foldRight de la clase

List . Es similar al que vimos en Scheme: un método de plegado, que recorre una lista de derecha a izquierda, aplicando una función que transforma los elementos de dos elementos en uno. Los dos elementos son el resultado anterior (o el caso base, cuando la lista es vacía), y el elemento de la lista.

El perfil de su definición es el siguiente (siendo *A* el tipo de los elementos de la lista):

def foldRight[B](z: B)(f: (A, B) => B): B

En la definición se utiliza un *currying* (lo veremos más adelante). No nos preocupemos ahora mismo los dos paréntesis, tenemos que verlos como una forma de pasar dos argumentos al método:

el caso base, de tipo *B* la función de plegado, que toma un objeto de la lista (de tipo *A*) y el resultado del plegado anterior (de tipo *B*) y devuelve un valor de tipo *B*

Los tipos *A* y *B* pueden ser el mismo. Por ejemplo el siguiente plegado concatena todas las cadenas de una lista:

val lista1 = List("hola", "que" , "tal")

lista.foldRight("")(\_ + \_)

⇒

String = holaquetal

Y el siguiente ejemplo muestra un ejemplo de plegado en el que los tipos *A* y *B* son distintos:

lista1.foldRight(0) ((cad,n) => {cad.length+n})

⇒

Int = 10

Otra forma de hacer lo mismo:

def sumaLongitud(cad:String,n:Int) = cad.length+n

lista.foldRight(0)(sumaLongitud \_)

⇒

Int = 10

# 5. Ámbitos

## 5.1. Reglas de evaluación de expresiones Scala con ámbitos

El funcionamiento de los ámbitos en Scala es igual al de Scheme:

Una invocación a una función crea un nuevo ámbito en el que se evalúa el cuerpo de la función. Es el *ámbito de evaluación* de la función.

El ámbito de evaluación se crea dentro del ámbito en el que se definió la función a la que se invoca

Los argumentos de la función son variables locales de este nuevo ambito que quedan ligadas a los parámetros que se utilizan en la llamada.

En el nuevo ámbito se pueden definir variables locales.

En el nuevo ámbito se pueden obtener el valor de variables del ámbito padre.

### Primer ejemplo

Supongamos el siguiente código en Scala:

Se definen dos funciones f y g que definen cada una distintas variables locales y devuelven una suma de los parámetros con la variable local. En la última línea de código se realiza una invocación a f con los resultados devueltos por dos invocaciones a g .

def f(x: Int, y: Int): Int = {

val z = 5

x+y+z

}

def g(z: Int): Int = {

val x = 10

z+x

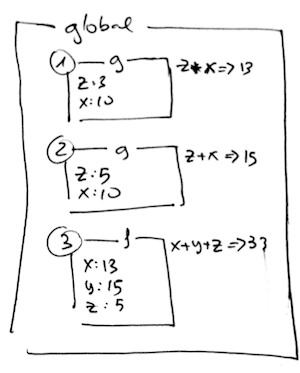
}

f(g(3),g(5))

¿Cuántos ámbitos locales se crean? ¿En qué orden?

1. En primer lugar se realizan las invocaciones a g . Cada una crea un ámbito local en el que se evalúa la función. Las invocaciones devuelven 13 y 15 respectivamente.
2. Después se realiza la invocación a f con esos valores 13 y 15. Esta invocación vuelve a crear un ámbito local en el que se evalúa la expresión ++y+z , devolviendo 33.

El diagrama de ámbitos locales generados por las sentencias anteriores es el siguiente:



## Segundo ejemplo con variables locales y globales

Supongamos el siguiente código en Scala:

val x = 10

val y = 20

def g(y: Int): Int = {

x+y

}

def prueba(z: Int): Int = {

val x = 0

g(x+y+z)

}

prueba(3)

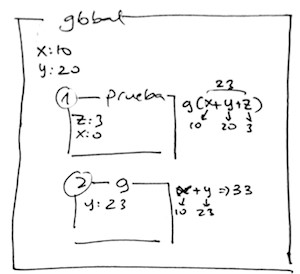
¿Qué devuelve prueba(3) ? ¿En qué ámbito se evalúa la expresión x+y+z ? ¿Y la expresión x+y ? ¿Qué valores tienen esas variables en el momento de la evaluación? Tenemos que aplicar

el modelo de ámbitos de Scala para descubrirlo.

En el caso del ejemplo, si ejecutamos el código en el intérprete veremos que devuelve 33. El funcionamiento es igual a Scheme:

1. En el ámbito principal del intérprete se define el valor de las variables x (10) e y (20), así como la función prueba .
2. La ejecución de prueba(3) crea un nuevo ámbito de evaluación, dentro del ámbito principal, en el que se ejecuta el cuerpo de la función.
3. En el ámbito de evaluación se liga el valor de z (argumento de prueba ) con el valor 3 (parámetro con el que se realiza la llamada).
4. En este nuevo ámbito se ejecuta el cuerpo de la función: se crea la variable local x con el valor 0 y se evalúa la expresión x+y+z . El valor de x y z están definidos en el propio ámbito local (0 y 3). El valor de y se obtiene del entorno padre: 20. El resultado de la expresión es 23.
5. Se invoca a la función g con el valor 23 como parámetro y . Para evaluar esta invocación se crea un nuevo ámbito local en el ámbito global en donde está definida la función.
6. Dentro de este nuevo ámbito se evalúa la expresión x+y devolviéndose 33 como resultado.

El diagrama de ámbitos que representa las invocaciones anteriores es el siguiente:



Por ahora es bastante normal. La diversión empieza cuando utilizamos la característica de que Scala (al igual que Scheme) puede construir funciones anónimas en la ejecución de otras funciones. Esto nos permite la definición de clausuras.

## 5.2. Clausuras en Scala

Siguiendo con el tema del funcionamiento de los ámbitos, ¿qué pasaría en el ejemplo anterior si prueba no evualúa ninguna expresión sino que devuelve una función anónima? Veámoslo, con una versión algo distinta del ejemplo anterior:

val x = 10

val y = 20

def prueba(y: Int): (Int)=>Int = {

val x = 0

(z:Int) => {x+y+z}

}

Hemos cambiado la definición de

prueba

haciendo que tome como parámetro la variable

y

y

que devuelva una función anónima construida en tiempo de ejecución: la función

}

(

z:Int) => {x+y+z

que tiene como parámetro

z

y como variables libres

x

e

y

. La

función es una clausura que se creará en al ámbito de evaluación de

prueba

y quedará

asociada a ese ámbito en el momento en que se ejecute

prueba

.

Supongamos que ejecutamos el siguiente código:

val f: (Int) => Int = prueba(10)

f(30)

Aunque no es necesario indicar el tipo de

f

(el compilador de Scala lo infiere) lo hacemos en el

ejemplo, para que se entienda mejor qué devuelve

prueba

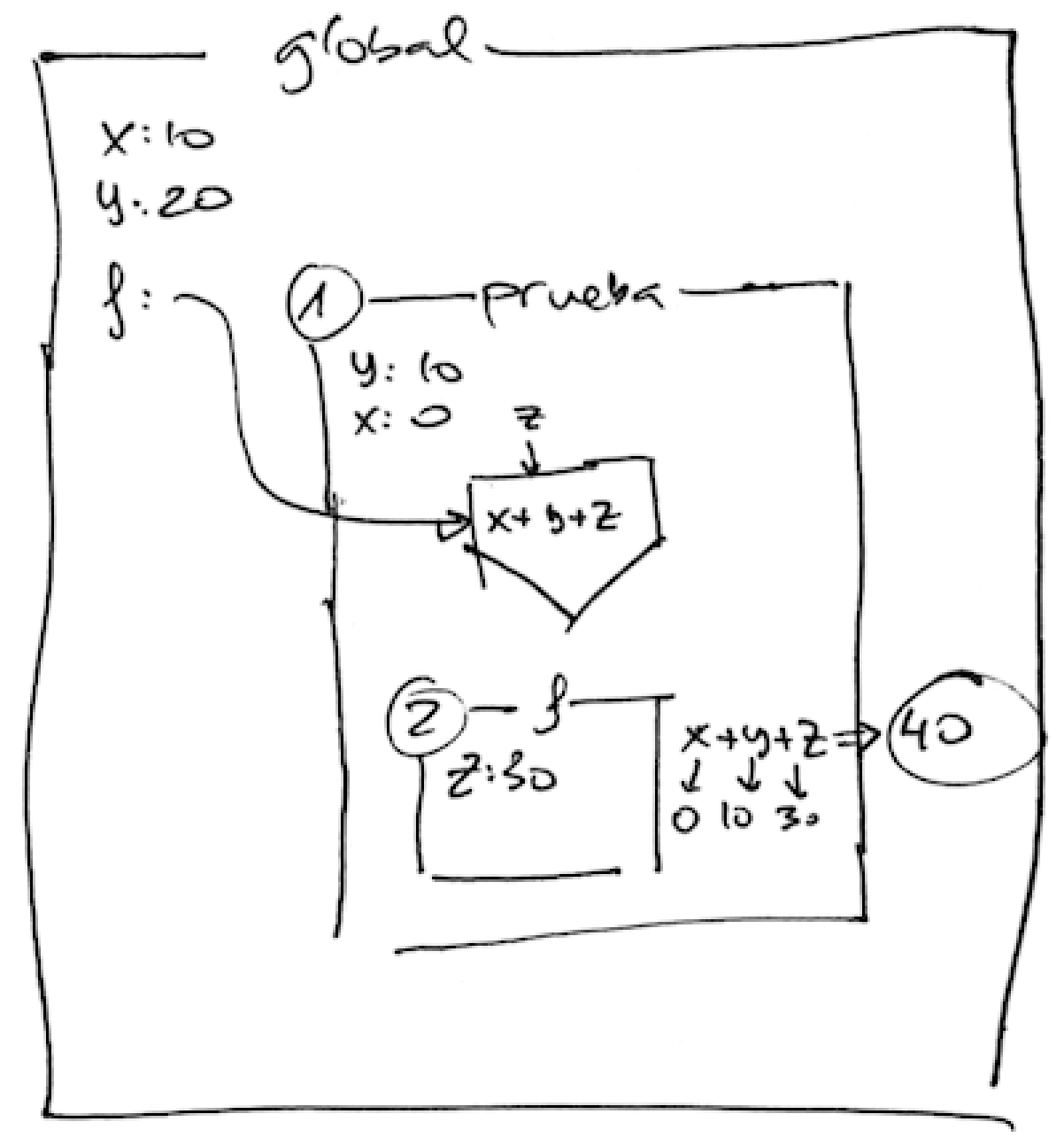
. El ejemplo funciona de la siguiente

forma:

1. Se invoca prueba(10) y se crea su ámbito de evaluación (dentro del ámbito global) en el que se definen las variables locales x e y con los valores 0 y 10.
2. En este ámbito se crea la función anónima (z:Int) => {x+y+z} . Se devuelve una referencia a esa función que se guarda en la variable f . La función anónima permanece en el ámbito de evaluación de prueba, junto con las variables locales. Es una clausura.
3. La invocación a f crea un nuevo ámbito de evaluación, dentro del de prueba y allí se evalua la expresión x+y+z . El valor de z es 30 (valor asignado al argumento en la llamada a f . El valor de x e y no está definido en el ámbito, pero sí en su padre: 0 y 10.

El valor resultante de la expresión es 40.

El diagrama de ámbitos que ilustra este comportamiento es el siguiente:



Algunos ejemplos de clausuras, similares a las que vimos en Scheme:

def makeSumador(k: Int): (Int)=>Int = (x: Int) => x + k

val f = makeSumador(10)

val g = makeSumador(100)

f(4)

g(4)

Y un último ejemplo

val x = 100

def func1(): () => Int = {

val x=1

()

=>

{x+1}

}

def func2(): Int= {

val x=10

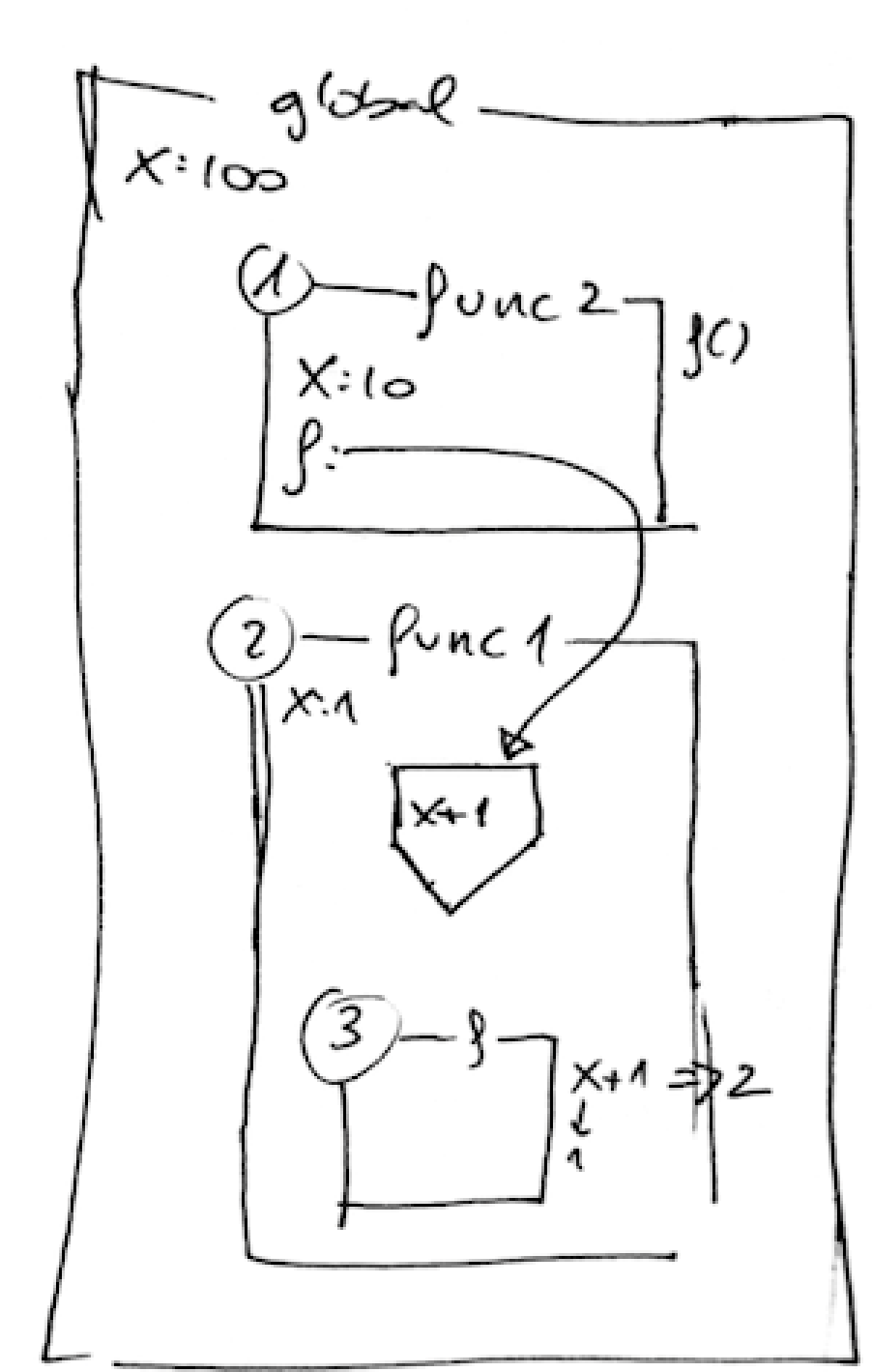
val f=func1()

f()

}

func2()

La llamada a func2 hace que se evalue su cuerpo. En él se crea la variable x con el valor 10 y la expresión f=func1() asocia f con la clausura devuelta por func1 . Esta clausura es



()=>{x+1} creada en el ámbito de evaluación de func1 . En ese mismo ámbito se ha creado también la variable x con el valor 1. La llamada a f en el código equivalente hace que la función se evalue.

El ámbito de evaluación de f se crea en el ámbito de evaluación de func1 , en el que x vale

1. El resultado de la llamada será 2.

Una formulación equivalente de

func2

es:

def func2(): Int= {

val x=10

func1()()

}

La llamada a

func1()

devuelve otra función que es evaluada con los siguientes paréntesis.

El código equivalente en Scheme sería:

(

define x

100)

(

define (func

1)

(let ((x 1))

(lambda () (+ x 1))))

(

define (func

2)

(let ((x 10))

((func1))))

(

func

2)

# 6. Otras características funcionales

## 6.1. Definición por comprensión de colecciones

Matemáticamente, una definición por comprensión de un conjunto se realiza con una definición lógica de una propiedad que deben satisfacer sus elementos. Por ejemplo, veamos la siguiente definición:

S = { x | x

∈

N : x == x

²

}

La definición se lee de la siguiente forma:

S es el conjunto de números con dominio en los números naturales (N) que tienen la propiedad de que el número coincide con el número al cuadrado.

El conjunto de números que cumple esta propiedad es {0, 1}.

Otro ejemplo es el conjunto de los números pares, que podríamos definir de la siguiente forma:

Pares = { x | x

∈

N : x % 2 == 0 }

Muchos lenguajes de programación funcional permiten definir listas utilizando propiedades similares. Las listas tienen que tener un número finito de elementos, por lo que no podemos hacer una definición tan general como la anterior. En su lugar podemos hacer definiciones como la siguiente:

{

x

²

| x

∈

{1..5} }

Estamos definiendo una lista con los cinco primeros números naturales elevados al cuadrado. Es muy sencillo expresar esta lista en Scala:

for (x <- (1 to 5)) yield x\*x

⇒

Vector(1, 4, 9, 16, 25)

La expresión

(1

to

5)

devuelve un

*rango*

de números del 1 al 5:

Un rango es un tipo particular de colección de Scala. La sentencia for … yield aplica la función definida tras el yield a todos los elementos de la colección sobre la que itera x y devuelve otra colección transformada con la expresión. En este caso, un vector con los números 1 al 5 elevados al cuadrado.

(1

to

5)

⇒

Range(1, 2, 3, 4, 5)

El operador

yield

también se puede aplicar a listas. En este caso devuelve una lista en lugar de

un vector. Ambos tipos de datos son colecciones.

for (x <- List.range(1,6)) yield x\*x

⇒

List(1, 4, 9, 16, 25)

Es posible también definir una condición que deben cumplir los números, antes de aplicar la expresión que va generando la lista final. Por ejemplo, la siguiente expresión devuelve el cuadrado de los números impares del 1 al 100: for (i <- List.range(1, 30) if i % 2 != 0) yield i\*i

⇒ List[Int] = List(1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841)

También es posible definir condiciones más complicadas. Por ejemplo, la siguiente expresión devuelve el cuadrado de los números impares divisibles por 3: for (i <- List.range(1, 101) if (i % 2 != 0 && i % 3 == 0)) yield i\*i

⇒List[Int] = List(9, 81, 225, 441, 729)

Y también es posible anidar varios for recorriendo más de una colección para tomar los distintos elementos que generarán la colección final. En el siguiente ejemplo se devuelve una colección de parejas formadas con dos generadores. El primero (1 to 3) define los primeros números de las parejas y el segundo (1 to x) utiliza la variable del primer generador como tope para obtener los valores de los segundos números: for(x <- (1 to 3 ); y <- (1 to x)) yield (x,y) ⇒ Vector((1,1), (2,1), (2,2), (3,1), (3,2), (3,3))

Una nota muy importante: aunque el bucle for es tipicamente procedural, en las definiciones anteriores por comprensión de Scala se utiliza de forma funcional. La sentencia no tiene efectos laterales y devuelve una colección. Es posible implementar sentencias similares de forma recursiva.

## 6.2. *Currying*

*Currying* es el proceso de convertir una función que toma *n* argumentos en funciones que toman un argumento. Cada función devuelve otra función que consume el segundo argumento.

Veamos un ejemplo. La siguiente función toma un argumento x , devolviendo a su vez otra función que toma otro argumento y y que evalúa la expresión x \* y :

def mulUnoCadaVez(x: Int) = (y: Int) => x \* y

Para multiplicar dos números, se puede invocar la función de la siguiente forma:

El resultado de mulUnoCadaVez(6) es la clausura (y: Int) => x \* y con el valor x asociado a 6. Esta función se aplica a 7, devolviendo 42.

mulUnoCadaVez(6)(7)

⇒

Int = 42

En Scala existe una forma abreviada de definir estas funciones *currificadas*:

def mulUnoCadaVez(x: Int)(y: Int) = x \* y;

## Bibliografía



Martin Odersky: “Programming in Scala”

Cay S. Horstmann:

“Scala for the impatient”

, Addison-Wesley Professional, 2012.

Martin Odersky,

“A Postfunctional Language”

Mario Gleichman, Serie de posts

“Functional Scala”

Lenguajes y Paradigmas de Programación, curso 2013–14

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante

Domingo Gallardo, Cristina Pomares