Programación Funcional

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-20

Objetivos

Objetivos

Los objetivos de este tema son:

- Presentar los conceptos de la programación funcional.
- Repasar las características básicas de la PF.
- Ofrecer algunos ejemplos de programación.

Introducción

Paradigma Imperativo

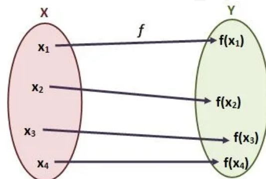
Los conceptos principales usados en los **lenguajes imperativos** son:

- **variable**: describe el estado del programa.
  Pueden representar direcciones (*l–value*) o valores (*r–value*).
- **asignación**: modificación de los valores de las variables
- **secuencia**: el significado (resultado) depende del orden de ejecución.
- **iteración**: uso extensivo de bucles

Debido a estas características a los lenguajes imperativos también se les denomina *lenguajes basados en **estados*** o *lenguajes orientados a la **asignación***.

Una función matemática es una aplicación de los miembros de un conjunto, denominado conjunto *dominio*, en los miembros de otro conjunto, denominado conjunto *rango*, de la manera siguiente:

A cada uno de los miembros del dominio le corresponde un miembro en el rango



El dominio y el rango pueden ser finitos o infinitos, iguales o distintos.

Una función matemática puede representarse de distintas formas:

- enumeración de los pares [dominio,rango]
- mediante una expresión que indica como calcular el valor

Ejemplos (notación matemática):

$$plusone(x) = x + 1 \tag{1}$$

$$square(x) = x * x \tag{2}$$

$$absolute(x) = \begin{cases} x < 0, & -x \\ x >= 0, & x \end{cases} \tag{3}$$

$$factorial(n) = \begin{cases} n = 0, & 1 \\ n > 0, & n * factorial(n-1) \end{cases} \tag{4}$$

Funciones matemáticas: Representación

Una función matemática puede representarse de distintas formas:

- enumeración de los pares [dominio,rango]
- mediante una expresión que indica como calcular el valor

Ejemplos (notación algorítmica):

```
  plusone[x]  ::=  x + 1
   square[x]  ::=  x * x
 absolute[x]  ::=  if x<0  then -x
                           else x
factorial[n]  ::=  if n=0  then 1
                           else n*factorial[n-1]
```

Si lo comparamos el concepto de función en programación imperativa con las funciones matemáticas, éstas presentan algunas diferencias:

- El concepto de variable está asociado al de valor y una vez que a una variable se le asocia un valor, éste no cambia.
- El valor de una función no depende de cosas como el orden de ejecución, o información externa a la propia defininción de la función (*global*).
- Las decisiones se implementan mediante **expresiones** condicionales.
- Se utiliza la recursión en lugar de la iteración.

En Matemáticas:

- El orden de evaluación de las funciones se controla mediante **expresiones** condicionales y **recursión**, en vez de con la secuenciación que ofrecen las instrucciones condicionales o iterativas.
- Las funciones matemáticas no producen efectos laterales. Por lo tanto con los mismos argumentos se generan los mismos resultados.
  Esta propiedad recibe el nombre de **Transparencia Referencial**.

La propiedad de la **Trasparencia Referencial** se cumple cuando el resultado de una función no depende de cuándo se llama a la función (orden de evaluación), sino de cómo se llama a esa función (sus argumentos).

Tomemos estas dos expresiones:

$$f(x) + f(x) \qquad\qquad 2 \cdot f(x)$$

Matemáticamente son equivalentes.
En un lenguaje de programación imperativo no podemos asegurar que sean equivalentes.
Su valor puede depender de otros factores además de los parámetros:

```
1  int k = 0;
2
3  int f(int x) {
4    k++;
5    return x + k;
6  }
```

Una función matemática puede observarse como si fuera un programa:

- tiene entradas
- existen reglas para combinar esas entradas y obtener un resultado
- puede utilizar otras funciones (*composición*)
- su definición puede ser recursiva

Si se basa un lenguaje en funciones puras (al estilo de las funciones matemáticas) con las únicas operaciones de composición funcional, recursión y expresiones condicionales se puede obtener un sistema poderoso, expresivo y semánticamente elegante.

La **Programación Funcional** es un paradigma de programación que trata los cálculos y los algoritmos como la evaluación de funciones matemáticas.

Evita los conceptos de *estado* del programa y *mutabilidad*.

Utiliza la evaluación de funciones en lugar de la modificación del estado del programa: no existen las variables (al estilo de la programación imperativa).

Un lenguaje funcional ofrece como base:

- un mecanismo de aplicación de funciones (*evaluación*)
- un conjunto de funciones primitivas
- un conjunto de *formas funcionales* para combinarlas (*composición*)
- algún tipo de estructura de datos para almacenar información (vectores, listas, tuplas, etc. . .)

El **Cálculo Lambda** (lambda–cálculo o λ–cálculo) es un sistema formal en lógica matemática para expresar cálculos computacionales y algoritmos mediante la asociación de variables y la sustitución.

Aunque es una abstracción matemática, sirve como base de todos los lenguajes funcionales modernos.

Características

En los lenguajes funcionales, las funciones son

objetos de primera clase

Eso significa que, aparte de poderse evaluar:

- existen funciones anónimas e *inline*

```
float x  <->  3.14
  sqr()  <->  (\x -> x*x)
```

- pueden pasarse como argumentos

```
draw (\x -> 3*x + 1)
```

- pueden devolverse como resultado de otras funciones
- se pueden almacenar en estructuras de datos

En los lenguajes funcionales, existen funciones anónimas e *inline*.

```
1 ghci> v   = 2
2 ghci> f x = 2*x+1
3 ghci> g   = (\x -> 2*x+1)
4 ghci> f v
5 5
6 ghci> g v
7 5
8 ghci> (\x -> 2*x+1) 3
9 7
```

En los lenguajes funcionales, las funciones pueden pasarse como argumentos y devolverse como resultado.

Ejemplo: composición

```
ghci> odd 4
False
ghci> even = not . odd
ghci> even 4
True
```

En los lenguajes funcionales, las funciones y los operadores son conceptos intercambiables.

```
1  ghci>  1 + 2
2  3
3  ghci> (+) 1 2
4  3
5  ghci> mod 8 3
6  2
7  ghci> 8 `mod` 3
8  2
```

No existe el concepto de variables globales.

En los lenguajes funcionales, las funciones son **puras**: no tienen ningún tipo de efecto lateral, ni en memoria, ni en IO, etc...

Como consecuencia:

- si su resultado no se usa, puede eliminarse su llamada
- si se llama con los mismos paramétros, da el mismo resultado
- si no hay dependencias visibles entre dos expresiones, pueden evaluarse en cualquier orden
- se puede utilizar cualquier *estrategia de evaluación* (...)

Las funciones que operan con otras funciones se denominan **funciones de orden superior** o **formas funcionales**.

Ejemplo: `map`

```
1 ghci> f = (\x -> 3*x+1)
2 ghci> map f [1, 2, 3, 4, 5]
3 [4,7,10,13,16]
4
5 ghci> map (\x -> 3*x+1) [1, 2, 3, 4, 5]
6 [4,7,10,13,16]
```

La repetición se consigue por medio de la recursión en lugar de la iteración.

```
apply(f,src)
  local dst
  for (e : src)
    dst.push_back(f(e));
  return dst
```

```
apply f src =
  (f (head src)) : (apply f (tail src))
```

Debido al uso extensivo de la recursividad, en muchos lenguajes funcionales la estructura de datos fundamental es la **lista**.

Recursión

Al igual que en lenguajes imperativos (`foreach...`), se pueden extraer patrones típicos de recursividad en el tratamiento de las estructuras de datos, e implementarlos como funciones de orden superior:

- aplicación múltiple (`map`)
- selección o filtrado (`filter`)
- catamorfismo (reducciones, `fold`)
- anamorfismo (generadores, `unfold`)

Piensa qué sucede en C++ cuando se evalúa una expresión como la siguiente:

```
int g(int x) { ... }
int h(int x) { ... }
```

```
int f(int a,int b)
{
  if (a>0)
    return a;
  else
    return a+b;
}
```

```
f( g(1) , h(2) )
```

En particular,

- ¿ en qué orden se evalúan las llamadas a cada función ?
- ¿ por qué ?

Los lenguajes funcionales pueden utilizar distintas **estrategias de evaluación** de expresiones, en particular de llamadas a función:

- **aplicativa**
  los parámetros se evalúan antes de efectuar la llamada
- **en orden normal**
  los parámetros sólo se evalúan en el momento en que se usan, en caso de que se usen, y cada vez que se usan
- **perezosa**
  los parámetros sólo se evalúan la primera vez que se usan, en caso de que se usen, y luego su valor se reutiliza

¿ Cómo se evaluaría la expresión mediante cada una de las estrategias ?

```
int g(int x) { ... }
int h(int x) { ... }

int f(int a,int b)
{
  if (a>0)
    return a;
  else
    return a+b;
}
```

```
f( g(1) , h(2) ):

int f( a ≡ g(1), b ≡ h(2) )
{
  if (g(1)>0)
    return g(1);
  else
    return g(1)+h(2);
}
```

Ejemplo:

```
1 ghci> a = 2*3
2 ghci> length [2+1, 3*2, a, 5-4*a]
3 4
```

Ejemplo:

```
ghci> a = 2*3
ghci> length [2+1, 3*2, a, 5-4*a]
4
```

```
ghci> a = 1/0
ghci> length [2+1, 3*2, a, 5-4*a]
4
```

En consecuencia, las funciones se pueden clasificar como:

- **estrictas**: para evaluarla es necesario que todos sus argumentos se puedan evaluar.
- **no–estrictas**: puede ser que se puedan evaluar incluso si alguno de sus argumentos no se puedan evaluar.

Esa misma clasificación se puede aplicar al lenguaje completo, si la estrategia elegida por el lenguaje es única.

Las funciones con múltiples argumentos pueden verse como una cadena de funciones con un solo argumento.

Ejemplo:

```
add x y = x + y

add 3 4 -> (add 3) 4

succ = add 1
```

Ejemplos

hello.hs

```haskell
-- Hello World

main = putStrLn "Hello World!!"
```

$$factorial(n) = \begin{cases} n = 0, & 1 \\ n > 0, & n * factorial(n-1) \end{cases} \quad (5)$$

factorial.hs

```haskell
-- Factorial

main = print (factorial 10)

factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

Primes

primes.hs

```haskell
-- Primes

main = do
  putStrLn "Primos de 1 a 100"
  print (primes 100)

primes k = [n | n <- [2..k], all ((>0).rem n) [2..n-1]]
```

qsort.c

```c
void qsort(int a[], int lo, int hi)
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo; h = hi; p = a[hi];

    do {
      while ((l < h) && (a[l] <= p)) l = l+1;
      while ((h > l) && (a[h] >= p)) h = h-1;
      if (l < h) {
          t = a[l]; a[l] = a[h]; a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

qsort.hs

```haskell
qsort :: Ord a => [a] -> [a]
qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort greater)
    where lesser  = filter (<  p) xs
          greater = filter (>= p) xs
```

qsort-2.hs

```haskell
qsort :: Ord a => [a] -> [a]
qsort (p:xs) = (qsort [x | x<-xs, x<p]) ++ [p] ++ (qsort [x | x<-xs, x>=p])
```

Qué opinas de la eficiencia de estas soluciones ?

Programación Funcional

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-20