Funciones abstractas

Juan Álvarez Rubio
jalvarez@dcc.uchile.cl

Las funciones para listas de datos recursivas presentan una forma o esquema similar que distingue el caso de una lista vacía del caso de una lista con al menos un valor:

```
from lista import *
#nombre: lista … -> …
def nombre(L,…):
  assert esLista(L)
  if vacia(L):
      return …
  else:
      …cabeza(L)…            #procesar primer valor de lista
      … nombre(cola(L),…) … #procesar recursivamente resto de lista
      return …
```

En esta sección se analizan tres procesos habituales que se realizan con los valores de una lista: la aplicación de una función, la selección de valores, y la reducción a un solo valor. Estos procesos se escribirán en funciones abstractas o genéricas de manera que puedan ser adaptadas a casos particulares.

Aplicación de una función a todos los valores de una lista

Una función puede aplicarse a todos los valores de una lista. Por ejemplo, para calcular la raíz cuadrada de todos los números de una lista se puede escribir la siguiente función:

```
from lista import *
from math import sqrt
L=lista(9,lista(25,None))
#raices: lista(num) -> lista(float)
#lista con raíces de números de lista L
#ej: raices(L) -> lista(3.0,lista(5.0,None))
def raices(L):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  else:
    return lista(sqrt(cabeza(L)),raices(cola(L)))
assert raices(L)==lista(3.0,lista(5.0,None))
```

Una situación similar se presenta en una función que recibe una lista de strings y entrega una lista con los largos o cantidad de caracteres de cada string:

```
from lista import *
L=lista('ana',lista('juan',None))
#largos: lista(str) -> lista(int)
#lista con largos de strings de lista L
#ej: largos(L)->lista(3,lista(4,None))
def largos(L):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  else:
    return lista(len(cabeza(L)),largos(cola(L)))
assert largos(L) == lista(3,lista(4,None))
```

En este caso se aplica la función predefinida len que recibe un string y entrega la cantidad de caracteres que contiene. Las funciones raíces y largos son similares y solo se diferencian en la función que aplican a cada valor de la lista: sqrt en el primer caso y len en el segundo. Por lo tanto, se puede escribir una función abstracta o genérica que reciba como parámetros una lista L y una función f que debe aplicarse a cada valor de la lista. Esta función, llamada mapa, contiene las siguientes instrucciones:

```
from lista import *
#mapa: lista(X) (X->Y) -> lista(Y)
#lista con f(valor1),f(valor2),…
def mapa(L,f):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  else:
    return lista(f(cabeza(L)), mapa(cola(L),f))
```

El contrato de la función mapa indica que recibe dos parámetros: una lista de valores del tipo X y una función que recibe un valor de tipo X y entrega un resultado de tipo Y. El resultado será una lista con valores de tipo Y. Eventualmente los tipos X e Y pueden ser iguales. La función abstracta mapa se puede usar para resolver los problemas de las raíces cuadrados y el de los largos de los strings de la siguiente manera:

```
from funcionesAbstractas import mapa
from math import sqrt
def raices(L):
  return mapa(L,sqrt)
def largos(L):
  return mapa(L,len)
```

La función mapa, que forma parte del módulo funcionesAbstractas, puede ser invocada también con funciones definidas por el programador. Por ejemplo, para sumar 1 a todos los números de una lista L se puede escribir:

```
def incrementar(x): return x+1
L1=mapa(L,incrementar)
```

Para funciones internas que se invocan una sola vez y que evalúan una expresión, Python permite la notación más breve: `L1=mapa(L,lambda x: x+1)`.

La notación "**`lambda parámetros: expresión`**" permite definir funciones anónimas (sin nombre) para ser utilizadas por una sola vez. El significado indica que es equivalente a escribir el nombre de una función definida como `def nombre(parámetros): return expresión`.

Seleccionar o filtrar valores de una lista

Una situación muy habitual es seleccionar los valores de una lista que cumplan una condición. Por ejemplo, la siguiente función entrega una lista con los primos de una lista de números:

```
from lista import *
from primos import esPrimo
L=lista(5,lista(4,lista(7,None)))
#listaPrimos: lista(int) -> lista(int)
#lista con primos de L
#ej: listaPrimos(L)->lista(5,lista(7,None))
def listaPrimos(L):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  elif esPrimo(cabeza(L)):
    return lista(cabeza(L),listaPrimos(cola(L)))
  else:
    return listaPrimos(cola(L))
assert listaPrimos(L)==lista(5,lista(7,None))
assert listaPrimos(listaVacia)==listaVacia
```

La función listaPrimos utiliza la función esPrimo (del módulo primos) que devuelve el valor True si su argumento es un número primo. La función recorre todos los valores de una lista y entrega otra lista que contiene solo los números primos de la lista original.

Una situación similar se presenta en el problema de seleccionar todos los números de una lista que son menores a un valor:

```
from lista import *
```

```
L=lista(5,lista(4,lista(7,None)))
#menores: lista(any) any -> lista(any)
#lista con valores de L menores que x
#ej: menores(L,6)->lista(5,lista(4,None))
def menores(L,x):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  elif cabeza(L) < x:
    return lista(cabeza(L),menores(cola(L),x))
  else:
    return menores(cola(L),x)
assert menores(L,6)==lista(5,lista(4,None))
assert menores(listaVacia,5)==listaVacia
```

Las funciones listaPrimos y menores siguen el mismo esquema, y solo se diferencian en la condición para seleccionar los valores de la lista original que se traspasan a la lista con los resultados. Por lo tanto, se puede escribir una función abstracta que reciba una lista y reciba la condición a través de una función que reciba un valor y entregue un resultado True o False. La función se denomina filtro porque filtra o selecciona los valores de una lista para los que la función devuelve un resultado True.

```
#filtro: lista(X) (X->bool) -> lista(X)
#lista con valores de L tales que f(valor) entrega True
def filtro(L,f):
  assert esLista(L)
  if vacia(L):
    return listaVacia
  elif f(cabeza(L)):
    return lista(cabeza(L),filtro(cola(L),f))
  else:
    return filtro(cola(L),f)
```

El contrato de la función filtro indica que recibe dos parámetros: una lista de valores del tipo X y una función que recibe un valor de tipo X y entrega un resultado de tipo bool. El resultado será una lista con valores de tipo X. La función abstracta filtro se puede utilizar para resolver los problemas de seleccionar los primos y los menores a un valor de la siguiente manera:

```
from funcionesAbstractas import filtro
from primos import esPrimo
def listaPrimos(L):
  return filtro(L,esPrimo)
def menores(L,x):
  return filtro(L,lambda n: n<x)
```

Reducir una lista a un solo valor

Es habitual realizar una operación con todos los valores de una lista. Por ejemplo, para multiplicar todos los números de una lista se puede invocar a la siguiente función:

```
from lista import *
L=lista(5,lista(7,None))
#multiplicar: lista (num) -> num
#multiplicar números de L
#ejs: multiplicar(L)->35, multiplicar(listaVacia)->1
def multiplicar(L):
    assert esLista(L)
    if vacia(L):
        return 1
    else:
        return cabeza(L) * multiplicar(cola(L))
assert multiplicar(L)==35
assert multiplicar(listaVacia)==1
```

El operador + utilizado entre dos strings (operandos de tipo str) junta o concatena los dos strings. Por ejemplo, 'a'+'b' entrega como resultado el string 'ab'. Al respecto, se puede escribir una función para concatenar todos los strings de una lista:

```
from lista import *
#concatenar: lista(str) -> str
#valor1 + valor2 + ...
#ej: concatenar(lista('a',lista('b',None))) -> 'ab'
#    concatenar(listaVacia) -> ''
def concatenar(L):
    assert esLista(L)
    if vacia(L):
        return ''
    else:
        return cabeza(L) + concatenar(cola(L))
assert  concatenar(lista('a',lista('b',None))) == 'ab'
assert concatenar(listaVacia) == ''
```

Las funciones multiplicar y concatenar son muy similares, y solo se diferencian en la operación que realizan y el valor que retornan en el caso de una lista vacía. En consecuencia, se puede escribir una función abstracta que reciba una lista, una función con la operación a realizar, y un valor que se entrega en caso de una lista vacía. Este último parámetro recibe inicialmente al valor neutro para la operación y se utiliza recursivamente para acumular el resultado. La función abstracta se denomina reductor porque reduce todos los valores de la lista a un solo valor.

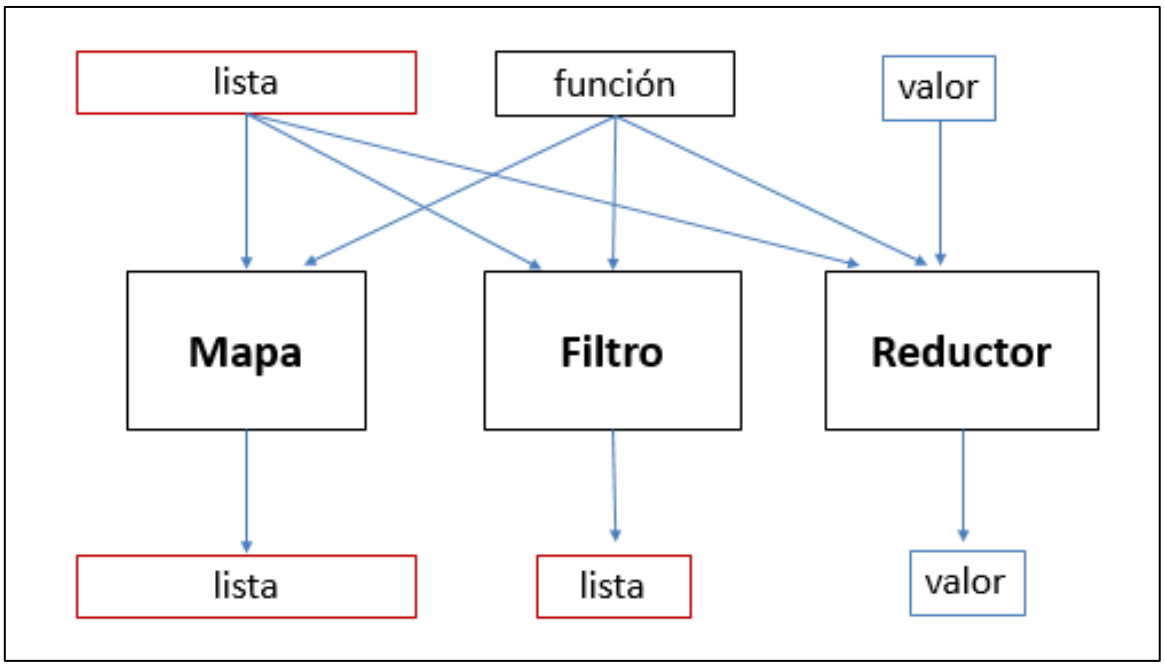
```
#reductor:lista(X) (X X->X) X -> X
#f(...f(f(valor,valor1),valor2)...)
def reductor(L,f,valor):
  assert esLista(L)
  if vacia(L):
      return valor
  else:
      resultado= f(valor,cabeza(L))
      return reductor(cola(L),f,resultado)
```

El contrato de la función reductor especifica que recibe tres parámetros: una lista de valores del tipo X, una función que recibe un valor de tipo X y entrega un resultado de tipo X, y un valor de tipo X. El resultado será un valor de tipo X. La función abstracta reductor se puede utilizar para resolver los problemas de multiplicar y concatenar de la siguiente manera:

```
from funcionesAbstractas import reductor
def multiplicar(L):
  return reductor(L, lambda x,y: x*y, 1)
def concatenar(L):
  return reductor(L, lambda x,y: x+y, '')
```

Resumen de las funciones abstractas

La siguiente figura ilustra los parámetros y los resultados de las funciones abstractas mapa, filtro y reductor. Las tres funciones reciben como parámetros una lista y una función (y la función reductor recibe un valor como tercer parámetro). Respecto de los resultados, la función mapa entrega una lista del mismo tamaño de la original, la función filtro entrega una lista más pequeña (eventualmente vacía o a lo más del mismo tamaño de la original), y la función reductor devuelve un solo valor como resultado.



Las tres funciones abstractas componen un módulo de manera que las funciones o programas que las necesiten puedan importarlas. Las instrucciones del módulo son las siguientes:

```
#modulo funcionesAbstractas
from lista import *

#mapa: lista(X) (X->Y) -> lista(Y)
#lista con f(valor1),f(valor2), . . .
#ej: mapa(lista(1,lista(2,None)),lambda x:x+1)
#       ->lista(2,lista(3,None))
def mapa(L,f):
  assert esLista(L)
  if vacia(L):
      return listaVacia
  else:
      return lista(f(cabeza(L)),mapa(cola(L),f))
assert mapa(lista(1,lista(2,None)),lambda x:x+1)\
         == lista(2,lista(3,None))

#filtro: lista(X) (X->bool) -> lista(X)
#lista con valores de L tales que f(valor) es True
#ej: filtro(lista(1,lista(2,None)),lambda x:x>1)->lista(2,None)
def filtro(L,f):
  assert esLista(L)
  if vacia(L):
      return listaVacia
  elif f(cabeza(L)):
```

```
    return lista(cabeza(L),filtro(cola(L),f))
  else:
    return filtro(cola(L),f)
assert filtro(lista(1,lista(2,None)),lambda x:x>1)==lista(2,None)

#reductor:lista(X) (X X->X) X -> X
#f(...f(f(valor,valor1),valor2))...)
#ej: reductor(lista(1,lista(2,None)),lambda x,y: x+y,0)->3
def reductor(L,f,valor):
  assert esLista(L)
  if vacia(L):
      return valor
  else:
      resultado= f(valor,cabeza(L))
      return reductor(cola(L),f,resultado)
assert reductor(lista(1,lista(2,None)),lambda x,y: x+y,0)==3
```

Ordenar una lista usando funciones abstractas

Las funciones abstractas pueden ser utilizadas para resolver problemas. Por ejemplo, se pueden utilizar en una función que reciba una lista y entregue otra con los valores ordenados de menor a mayor. La solución más intuitiva sugiere que se entregue como resultado una lista que tenga como primer valor el menor de los valores de la lista original. Los siguientes valores se obtienen ordenando recursivamente la lista pero eliminado el menor valor. Para obtener el menor valor se puede utilizar un reductor y para eliminarlo se puede usar un filtro que seleccione los valores mayores al menor:

```
from funcionesAbstractas import *
L=lista(2,lista(3,lista(1,None)))
#ordenar: lista -> lista
#lista con valores de L ordenados (y distintos)
#ej: ordenar(L)->lista(1,lista(2,lista(3,None)))
def ordenar(L):
    assert esLista(L)
    if vacia(L): return listaVacia
    menor=reductor(L,min,cabeza(L))
    mayores=filtro(L,lambda x: x>menor)
    return lista(menor,ordenar(mayores))
assert ordenar(L)== lista(1,lista(2,lista(3,None)))
```

El reductor recibe la función predefinida min para obtener el menor entre dos valores, por lo tanto, puede servir para listas de números y de strings. Por otra parte, el argumento inicial para el tercer parámetro del reductor es el primer valor de la lista. Consecuentemente, el primer argumento para el reductor podría ser también cola(L), es decir, la lista desde el segundo valor en adelante.

La función anterior supone que la lista no tiene valores repetidos. Esto significa que el resultado que entrega el reductor se utiliza para generar la lista con los mayores que el menor. Si la lista presenta valores repetidos entonces la lista con el resultado se debe obtener concatenando ("pegando") la lista de todos los iguales al menor con la lista ordenada de los mayores:

```
L=lista(2,lista(3,lista(1,lista(3,None))))
#ordenar: lista -> lista
#lista con valores de L ordenados
#ej: ordenar(L)->lista(1,lista(2,lista(3,lista(3,None))))
def ordenar(L):
    assert esLista(L)
    if vacia(L): return listaVacia
    menor=reductor(cola(L),min,cabeza(L))
    menores=filtro(L,lambda x: x==menor)
    mayores=filtro(L,lambda x: x>menor)
    def concatenar(x,y):
        if vacia(x): return y
        return lista(cabeza(x),concatenar(cola(x),y))
    return concatenar(menores,ordenar(mayores))
assert ordenar(L)==lista(1,lista(2,lista(3,lista(3,None))))
```

Las funciones abstractas se pueden utilizar también con un tipo de datos definidos por el programador. Por ejemplo, para ordenar una lista de fracciones se puede escribir la siguiente función que utiliza el módulo fraccion:

```
from funcionesAbstractas import *
from fraccion import *
L=lista(fraccion(2,3),lista(fraccion(1,2),None))

#ordenarFracciones: lista(fraccion) -> lista(fraccion)
#lista con valores de L ordenados
#ej: ordenarFracciones(L)->
#    lista(fraccion(1,2),lista(fraccion(2,3),None))
def ordenarFracciones(L):
    assert esLista(L)
    if vacia(L): return listaVacia
    def min(x,y): return x if comparar(x,y)<0 else y
    menor=reductor(cola(L),min,cabeza(L))
    menores=filtro(L,lambda x: comparar(x,menor)==0)
    mayores=filtro(L,lambda x: comparar(x,menor)>0)
    def concatenar(x,y):
        if vacia(x): return y
        return lista(cabeza(x),concatenar(cola(x),y))
    return concatenar(menores,ordenarFracciones(mayores))
```

```
assert ordenarFracciones(L)==\
       lista(fraccion(1,2),lista(fraccion(2,3),None))
```

La expresión “x if comparar(x,y)<0 else y” se escribe en una línea y responde a la forma general “expresión1 if condición else expresión2” que entrega por resultado el valor de la expresión1 o de la expresión2 si la condición es True o False respectivamente. Por lo tanto, la función min es equivalente a:

```
def min(x,y):
  if comparar(x,y)<0:
    return x
  else:
    return y
```