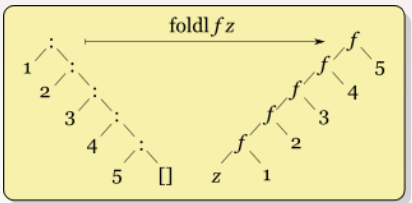
04834580 Software Engineering (Honor Track) 2024-25

Functional Programming Patterns

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

The `reduce` (or `foldl`) function takes a list of values and reduces it to a single value.

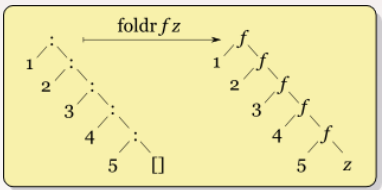
```
from functools import reduce
f = lambda x, y: x * y
product = reduce(f, [1, 2, 3, 4, 5])
```




```
foldl = lambda func, acc, xs: reduce(func, xs, acc)
>>> foldl(operator.sub, 0, [1,2,3])
-6
>>> foldl(operator.add, 'L', ['1','2','3'])
'L123'
```

Foldr Function

The foldr function applies a function against an accumulator and each value of a foldable structure from right to left, folding it to a single value.




```
import functools
import operator
foldr = lambda func, acc, xs: reduce(lambda x, y: func(y, x), xs[::-1], acc)
>>> foldr(operator.sub, 0, [1,2,3])
2
>>> foldr(operator.add, 'R', ['1', '2', '3'])
'123R'
```

Note: `xs[::-1]` is the Python idiomatic way to return the reverse of a list.

Expressiveness of Folds

Finding max and min elements, computing the product:

```
lmax = lambda xs: reduce(lambda x, y: x if x > y else y, xs)
>>> lmax([1,2,3,4,5])
5
lmin = lambda xs: reduce(lambda x, y: x if x < y else y, xs)
>>> lmin([1,2,3,4,5])
1
lsum = lambda xs: reduce(operator.add, xs)
>>> lsum([1,2,3,4,5])
15
product = lambda xs: reduce(operator.mul, xs)
>>> product([1,2,3,4,5])
120
```

Expressiveness of Folds

Predicates on list elements:

```
lany = lambda pred, xs: reduce(lambda x, y: x or pred(y), xs, False)
>>> lany(lambda x: x > 3, [1,2])
False
>>> lany(lambda x: x > 3, [1,2,3,4,5,6])
True
lall = lambda pred, xs: reduce(lambda x, y: x and pred(y), xs, True)
>>> lall(lambda x: x > 3, [4,5,6,7])
True
>>> lall(lambda x: x > 3, [1,2])
```

Expressiveness of Folds

Defining other high-level functions:

```
lmap = lambda func, xs: reduce(lambda x, y: x + [func(y)], xs, [])
>>> lmap(lambda x: x + 2, [1,2,3,4,5])
[3, 4, 5, 6, 7]
lfilter = lambda func, xs: reduce(lambda x, y: x + [y] if func(y) else x, xs, [])
>>> lfilter(lambda x: x % 2 == 0, [1,2,3,4,5,6,7,8,9])
[2, 4, 6, 8]
```

Insertion fold algorithm:

```
def insert(y, xs):
    if not xs:
        return [y]
    x, *rest = xs
    if y < x:
        return [y] + [x] + rest
    else:
        return [x] + insert(y, rest)

isort = lambda xs: foldl(insert, [], xs)
```

Arithmetic expressions:

```
class Num:
    def __init__(self, value):
        self.value = value

class Add:
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Sub:
    def __init__(self, left, right):
        self.left = left
        self.right = right
```

Traverse the tree recursively and apply the function `func` to combine the results of traversing each subtree:

```
def reduce_tree(fleaf, fadd, fsub, tree):
    if isinstance(tree, Num):
        return fleaf(tree.value)
    elif isinstance(tree, Add):
        return fadd(reduce_tree(fleaf, fadd, fsub, tree.left),
                    reduce_tree(fleaf, fadd, fsub, tree.right))
    elif isinstance(tree, Sub):
        return fadd(reduce_tree(fleaf, fadd, fsub, tree.left),
                    reduce_tree(fleaf, fadd, fsub, tree.right))
```

We can create a function to print the expression:

```
print_expr = lambda t: reduce_tree(str,
    lambda x, y: f"({x} + {y})",
    lambda x, y: f"({x} - {y})",
    t)
```

Folding Trees

We can also evaluate the expression using the fold function:

```
eval_expr = lambda t: reduce_tree(lambda x: x,
        lambda x, y: x + y,
        lambda x, y: x - y,
        t)
```

- A **Monad** is a design pattern used to represent computations chained together.
- It consists of two main components (in Haskell notation):
  1. **A type constructor**: It takes a type T and returns a new type $M[T]$.
  2. **Two operations**:
     - **bind**: `bind :: M[T] -> (T -> M[U]) -> M[U]`
     - **return** (or unit): `return :: T -> M[T]`

Example: Chaining Operations with Errors

北京大学

```python
from pymonad.either import Left, Right
from pymonad.tools import curry

def parse_int(value):
    try:
        return Right(int(value))
    except ValueError as e:
        return Left(f"Error parsing integer: {e}")

def square(value):
    return value ** 2

def string_template(value):
    return f"FINAL RESULT {value}"

result = Right('5').then(parse_int).then(square).then(string_template)

print(result)
```

Example: Chaining with Partial Application

北京大学

```python
from pymonad.either import Left, Right
from pymonad.tools import curry

@curry(2)
def divide(a, b):
    return (
        Right(a/b)
        if b != 0
        else Left("Error: Division by zero")
    )

result = Right(10).then(divide(2)).then(divide(5))

print(result)
```

Example: Encapsulating IO Operations

北京大学

```python
from pymonad.io import IO

def read_csv(filename):
    return IO(lambda: open(filename, 'r').read())


def process_csv(content):
    lines = content.split('\n')
    headers = lines[0].split(',')
    return IO(lambda: {'headers': headers,
                       'data': [line.split(',') for line in lines[1:]]})

filename = 'example.csv'
csv_program = read_csv(filename).then(process_csv)

result = csv_program.run()
print("CSV Headers:", result['headers'])
print("CSV Data:", result['data'])
```

Example: Monadic Logging

北京大学

```
@curry(2)
def add(x, y):
  return Writer(x + y,
    f"Called function 'add' with arguments {x} and {y}. Result: {x + y}")

@curry(2)
def mul(x, y):
  return Writer(x * y,
    f"Called function 'mul' with arguments {x} and {y}. Result: {x * y}")

logged_arithmetic = (Writer
                     .insert(0)
                     .then(add(1))
                     .then(mul(2)))
```

References I

[1] Jeremy Gibbons.
Origami programming.
2003.