Haskell. Функциональное программирование

Миша Чернигин, 22 февраля 2025 года

Haskell был создан в 1987 году и является чисто функциональным языком программирования, реализующий ленивые вычисления.

Функциональное программирование предпочитают партийные функционеры.

В функциональных языках нет переменных и мы не можем, например, делать циклический обход. Мы не можем изменять объекты, мы пишем функции, которые возвращают новые объекты.

Зачем? Рассмотрим пример.

Пусть у нас есть множество точек P, выберем точку $t \in P$ и попробуем найти ближайшую к ней точку. Императивное решение на питоне предполагает циклический перебор с вычислением расстояния и сохранением минимального. Эквивалент на хасекелле:

```
1 closest target = minimumBy (compare `on` (distance target))
```

Почему так получается? Хаскель делает упор не на написание кода, а на его чтение, поэтому идея хаскелля в создании наиболее удобочитаемого синтаксиса.

На питоне тоже можно сделать похожий код:

```
1 def closest(target, points):
2   return min(points, key = lambda p: distance(target, p))
```

“A language that doesn’t affect the way you think, is not worth knowing”

Создание функции двойного сложения выглядит так:

```
1 addTwice x y == add x y + add x y
```

Также хаскелль поддерживает инфиксную нотацию:

```
1 add x y == x `add` y
```

Списки реализуются с помощью оператора присоединения элемента к списку:

```
1 lst = 1 : (2 : (3 : nil))
```

`nil` в хаскелле — это не указатель в никуда, а обозначение пустого списка, эквивалентное двум скобкам.

Скобки хаскелль позволяет опускать:

```
1 lst = 1 : 2 : 3 : nil
```

Ветвление в хаскелле работает так:

```
1 f = if <condition> then <true_expr> else <false_expr>
```

Похожая конструкция есть и в C++, и в питоне (в обоих языках это тернарный оператор), но важно уточнить, что else является обязательным.

При описании функции можно делать сопоставление с образцом:

```
1 factorial 0 = 1
2 factorial n = n * factorial(n - 1)
```

Порядок важен!

Очень важно такую вещь использовать со списками:

```haskell
sum [] = 0
sum (x : xs) = x + sum xs
```

Подход сопоставления с образцом позволяет избавиться от циклов и реализовать рекурсивную функцию наиболее удобным способом.

```haskell
sign x | x > 0 = 1
       | x < 0 = -1
       | otherwise = 0
```

Для связывания внутри функций есть let и where:

```haskell
absDiff a b = let
  abs x | x < 0 = -x
        | otherwise = x
in
  abs a - abs b
```

Мы объявляем в блоке let..in функцию abs, которую мы используем внутри функции absDiff, но никогда за её пределами. Это нужно для объявлия локальных имён.

where — это обратная запись, которая используется чаще и менее понятна тем, кто пишет не на хаскелле:

```haskell
absDiff a b = abs a - abs b
  where
    abs x | x < 0 = -x
          | otherwise = x
```

Система типов и каррирование

Мы можем явно указывать поддерживаемые типы:

```haskell
add :: Integer -> (Integer -> Integer) -- скобки опциональны
add x y = x + y
```

Когда мы описываем функцию в хаскелле, мы описываем функцию, которая возвращает функцию от остальных элементов. Запись выше позволяет понять это. Последний тип — возвращаемое значение, а все предыдущие — типы аргументов.

Можно использовать что-то вроде шаблонов в C++:

```haskell
add :: a -> a -> a
add x y = x + y
```

Но этот код не рабочий, потому что сложение определено не для всех типов. Нужно ограничить разнообразие (*да ахуеет весь лгбт движ*), воспользовавшись чем-то похожим на интерфейсы — здесь оно называется классами.

```haskell
add :: Num a => a -> a -> a
add x y = x + y
```

Можно сделать так:

```haskell
add :: Num a => a -> a
add42 x = add 42
```

По сути то, что должно было быть y, стало константой.

Рассмотрим пример. Напишем на питоне решето Эратосфена для поиска простых чисел, который работает методом вычуркивания:

```python
def sieve_of_eratosthenes(n):
    # Создаем список булевых значений, изначально предполагаем, что все числа простые
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False  # 0 и 1 не являются простыми числами

    # Начинаем с первого простого числа — 2
    p = 2
    while p * p <= n:
        # Если p — простое число, то помечаем все его кратные как составные
        if is_prime[p]:
            for i in range(p * p, n + 1, p):
                is_prime[i] = False
        p += 1

    # Собираем все простые числа в список
    primes = [p for p in range(n + 1) if is_prime[p]]
    return primes

# Пример использования
n = 50
print(sieve_of_eratosthenes(n))
```

А теперь перепишем на хаскелль:

```haskell
primes = sieve[2..] -- список абсолютоно всех простых чисел
  where sieve(p : xs) = p : sieve (filter(\x ->x `mod` p /= 0) xs) -- /= --- это не равно

main = do
  print $ take 10 primes -- Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Разработчики PHP славятся своей жадностью, потому что они очень любят доллары. В хаскелле доллар — это синтаксический сахар для избегания скобок.

В лиспе любят скобки, в хаскелле — нет.

```haskell
print $ x == print(x)
print $ f $ x == print(f(x))
```

Определим собственный тип бинарного дерева:

```haskell
data Tree a = Empty
  | Node a (Tree a) (Tree a)
```

Empty, Node — типы, из которых мы конструируем дерево.

Подумаем над функцией поиска пути.

```haskell
findPathToLeaf :: Eq a -> s -> Tree a -> Maybe [a] -- Eq гарантирует возможность сравнения
findPathToLeaf _ Empty = Nothing -- Nothing - это конструктор опционального типа Maybe, который говорит, что результат не найден
findPathToLeaf target (Node value left right)
  | value == target && isLeaf (Noode value left right) = Just [value]
  | otherwise = case findPathToLeaf target left of
    Just path -> Just(value : path)
    Nothing -> case findPathToLeaf target right of
      Just path -> Just(value : path)
      Nothing -> Nothing
```

```haskell
isLeaf :: Tree a -> Bool
isLeaf (Node _ Empty Empty) = True
isLeaf _ = False
```

```haskell
main do
  let lhs = Node 2 Empty (Node 4 Empty Empty)
  let rhs = Node 3 (Node 5 Empty Empty) Empty
  let tree = Node 1 lhs rhs
print $ findPathToLeaf 4 tree
```

А код поиска в ширину будет выглядеть так:

```haskell
import Data.List (nub)

-- Определим тип графа как список смежности
type Graph = [(Int, [Int])]

-- Функция bfsSearch принимает граф, начальную вершину, целевую вершину и возвращает путь
bfsSearch :: Graph -> Int -> Int -> Maybe [Int]
bfsSearch graph start target = bfsHelper [(start, [start])] []
  where
    -- Вспомогательная функция для BFS
    bfsHelper [] _ = Nothing -- Если очередь пуста, путь не найден
    bfsHelper ((current, path):queue) visited =
      if current == target
        then Just (reverse path) -- Если достигли целевой вершины, возвращаем путь
        else
          if current `elem` visited
            then bfsHelper queue visited -- Если вершина уже посещена, пропускаем её
            else
              let neighbors = getNeighbors graph current -- Получаем соседей текущей вершины
                  newPaths = map (\n -> (n, n:path)) neighbors -- Создаем новые пути
                  newQueue = queue ++ newPaths -- Добавляем новые пути в очередь
                  newVisited = current : visited -- Добавляем текущую вершину в посещённые
              in bfsHelper newQueue newVisited

-- Функция для получения соседей вершины
getNeighbors :: Graph -> Int -> [Int]
getNeighbors graph node = case lookup node graph of
  Just neighbors -> neighbors
  Nothing -> []

-- Пример использования
main :: IO ()
main = do
  let graph = [(1, [2, 3]), (2, [4, 5]), (3, [6]), (4, []), (5, []), (6, [])]
  print $ bfsSearch graph 1 6
```

Производительность

О Производительности судить сложно, поскольку хаскелль более распространён в математических кругах, где не особо думают о реализации языках, но уделяют большое внимание синтаксису и семантике.

Многие функциональные подходы требуют из-за отсутствия изменяемости элементов их копирование, что сильно бьёт по эффективности готовой программы. Но в целом хаскелль достаточно быстрый, чтобы использовать его в решении задач с высокими требованиями к производительности.

Тот факт, что хаскелль — чисто функциональный язык, обеспечивает тривиальную реализацию параллелизма. Функциональные программы очень легко параллелизовать и выполнять на нескольких ядрах одновременно.

Монада — это потенциальная тема второй части этой лекции.