14.07.2023

Что такое функциональное программирование?

Долгое время у меня не было ответа, которым я был бы доволен. На протяжении многих лет я обнаружил, что это означает разные вещи для разных людей, но была явная общая идея, вокруг которой крутились более сложные идеи. В тот момент я остановился на "FP - это о функциях, их входах и выходах". Вот и все. Функции потребляют свои входы и производят выход. Все, что позволяет вам сделать функциональный язык программирования. В частности, здесь нет ничего лишнего. Только вход и выход.

Проблема в том, что трудно увидеть, как программу большего масштаба можно закодировать таким образом. Я бы сравнил это с утверждением, что эволюция связана с случайными мутациями. Да, это часть этого, но не вся история. Настолько, что это упускает главное. На протяжении долгих периодов времени в популяциях есть особи с мутацией, некоторые выживают, некоторые нет, и если эта мутация помогла особям выжить, то она, скорее всего, продолжит жить, поскольку потомки наследуют эту мутацию. Это гораздо лучшее описание эволюции. Вот что я искал, когда речь заходила о FP. Макроскопическое определение или описание, которое бы соответствовало моему повседневному опыту.

Затем, еще в 2020 году, я наткнулся на эту Game of life на JS. Вы могли видеть множество таких примеров на Go, Smalltalk (последняя страница) и APL. Мне показался удивительным код JS! Я никогда серьезно не рассматривал набор точек (x, y) (живых клеток) в качестве представления Жизни. Я всегда думал об этом и кодировал его как матрицу какого-то рода.

Вот код JS:

```
// See https://observablehq.com/@visnup/game-of-life
function tick(cells) {
  const counts = {}
  for (const cell of cells)
    for (const n of neighbors(cell.split(',').map(Number)))
      counts[n] = (counts[n] ?? 0) + 1

  const next = new Set()
  for (const [cell, count] of Object.entries(counts))
    if (count === 3 || (count == 2 && cells.has(cell)))
      next.add(cell)

  return next
}

function* neighbors([x, y]) {
  for (const dx of [-1, 0, 1])
    for (const dy of [-1, 0, 1])
      if (dx !== 0 || dy !== 0)
        yield [x + dx, y + dy]
}
```

Я приступил к воспроизведению этого на Elixir, Phoenix LiveView и SVG. Я начал с нуля, но мне не удалось сделать это так же аккуратно, как в коде JS. Поэтому я скопировал его в свой редактор и превратил в код Elixir, конструкцию за конструкцией, чтобы он как можно больше синтаксически напоминал оригинал на JS. Когда у меня все заработало, я начал перерабатывать его в нечто, что больше похоже на идиоматический функциональный код на Elixir (честно говоря, первая рабочая копия была довольно хороша). Я никогда бы не подумал, что перевод кода, конструкцию за конструкцией, с одного стиля на другой, будет полезным, но он оказался таковым. Я переработал код, и раскрутилась другая программа.

Я увидел, что то, что было вложенным блоком в JS, стало вложенными данными в Elixir, вложенные циклы for стали вложенными списками, а вложенная структура управления стала вложенной структурой данных. Переменные стали функциями. К концу я встроил программу в данные. Разница между структурированным программированием (код JS) и функциональным программированием (код Elixir) стала для меня такой ясной. Это легко видеть, если вы сравните и сопоставите JS выше с Elixir ниже построение за построением.

Вот итоговый код на Elixir:

```
defmodule Life do
  defmodule Cell,
    do: @type(t :: {x :: integer(), y :: integer()})

  defmodule World,
    do: @type(t :: [Cell.t()])

  @spec tick(World.t()) :: World.t()
  def tick(world) do
    world
    |> Enum.map(&neighbouring/1)
    |> List.flatten()
    |> Enum.frequencies()
    |> Enum.filter(alive?(world))
    |> Enum.map(fn {cell, _count} = _frequency -> cell end)
  end

  @spec alive?(World.t()) :: (frequency :: {Cell.t(), non_neg_integer()} -> boolean())
  defp alive?(world) do
    fn {cell, count} = _frequency ->
      count === 3 || (count === 2 && cell in world)
    end
  end

  @spec neighbouring(Cell.t()) :: [Cell.t()]
  defp neighbouring({x, y}) do
    for dx <- [-1, 0, 1], dy <- [-1, 0, 1], not (dx === 0 and dy === 0),
      do: {x + dx, y + dy}
  end
end
```

Теперь я предпочитаю использовать следующую императивную форму:

Манипулируйте данными программы; не манипулируйте текстом программы.

Суть FP не только в функциях, входах и выходах. FP - это о представлении; конкретно, об одном представлении за другим. Это последовательность данных; не последовательность инструкций. Это последовательность коллекций; не последовательность утверждений. Это ваше приложение снимок за снимком; не шаг за шагом. Теперь с этой идеей в голове гораздо легче понять, как большая программа кодируется в функциональном стиле, как она складывается из простых функций, их входов и выходов.

Итак, это история о том, как я нашел свой ответ. В заключение я хочу выделить три момента:

- Представление, как мы знаем, может сделать большую разницу. Так гласит изречение о том, что структуры данных важнее, чем функции, процедуры и т.д.
- Даже после многих лет просмотра проблемы, элегантность представления, приведенного выше, ускользала от меня. Мы можем пропустить отличное решение снова и снова. Возможно, стоит рассмотреть самые странные формулировки, используя все доступные нам инструменты, когда мы только начинаем.

- Перевод может быть ценным способом обучения. Я не думал, что это так, потому что это могло отвлечь от попадания в правильное мышление. Я ошибался.

Другие статьи, которые могут вам понравиться:

- An introduction to functional programming
- Designing with types: Making illegal states unrepresentable
- Parse, don’t validate