Функциональное программирование

Саймон Марлоу

Параллельное и конкурентное программирование на языке Haskell

ДМК издательство

УДК 004.432.42 Haskell
ББК 32.973.28-018.1
М28

М28 **Саймон Марлоу**
Параллельное и конкурентное программирование на языке Haskell / Пер. с англ. В. Н. Брагилевского. – М.: ДМК Пресс, 2014. — 372 с.: ил.
ISBN 978-5-94074-984-4

Если вы уже владеете программированием на языке Haskell, эта книга научит вас использованию множества интерфейсов и библиотек, предназначенных для написания параллельных и конкурентных программ. Вы узнаете, как распараллеливание на многоядерные процессоры позволяет ускорять вычислительно нагруженные программы и как конкурентность облегчает написание программ с активно взаимодействующими между собой и с другими программами потоками.

Автор Саймон Марлоу проведёт вас по этому пути, сопровождая его большим количеством примеров, с которыми можно самостоятельно экспериментировать, запуская, изменяя и расширяя. Книга делится на две части, посвящённые таким инструментам, как Parallel Haskell и Concurrent Haskell, включённые в неё упражнения позволят вам научиться:

- выражать параллелизм в языке Haskell средствами монады Eval и стратегий вычислений;
- распараллеливать обычный код на языке Haskell в монаде Par;
- организовывать параллельные вычисления с массивами на основе библиотеки Repa;
- использовать библиотеку Accelerate для запуска вычислений на графических процессорах;
- работать с базовыми интерфейсами для написания конкурентного кода;
- реализовывать высокопроизводительные конкурентные сетевые серверы;
- писать распределённые программы, запускающиеся на множестве машин сети.

УДК 004.432.42 Haskell
ББ 32.973.28-018.1

ISBN 978-1-449-33594-6 (англ.)
ISBN 978-5-94074-984-4 (рус.)

Оглавление

2. Простейший параллелизм: монада Eval

В этой главе рассказывается о простейших способах добавления параллелизма в код на языке Haskell. Мы начнём с базовых сведений о ленивых вычислениях, а затем, в разделе «Монада Eval, rpar и rseq» на с. 33, посмотрим, как использовать параллелизм.

Ленивые вычисления и слабая головная нормальная форма

Haskell — это *ленивый* язык, а значит, значения выражений не вычисляются до тех пор, пока они не потребуются[1]. Обычно об этом не приходится беспокоиться: поскольку выражения, как только их значения нам понадобятся, всё-таки вычисляются, и не вычисляются в противном случае, всё прекрасно. Однако при добавлении в код параллелизма мы начинаем указывать компилятору, как именно следует запускать программу: некоторые действия должны исполняться параллельно. В целях эффективного использования параллелизма стоит разобраться с тем, как именно работают ленивые вычисления, поэтому в этом разделе мы рассмотрим простейшие принципы, а в качестве песочницы воспользуемся интерпретатором GHCi.

Начнём с самого простого:

```
Prelude> let x = 1 + 2 :: Int
```

Здесь переменная `x` связывается с выражением `1+2` (типа `Int`, во избежание сложностей, связанных с перегрузкой). Далее, по крайней мере с точки зрения языка Haskell, `1+2` равно `3`. Можно было бы написать `let x = 3 :: Int`, и если писать обычный код на Haskell, то отличить эти два варианта невозможно. Однако мы говорим о параллелизме, поэтому

[1] Технически это неверно. На самом деле Haskell — это *нестрогий* язык, а ленивые вычисления — это один из возможных способов реализации. Но поскольку в Haskell действительно используются ленивые вычисления, эту тонкость мы пока будем игнорировать.

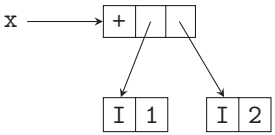

Рис. 2.1: задумка, представляющая сумму `1+2`

разница между `1+2` и `3` для нас существенна, потому что `1+2` — это пока невыполненное вычисление, а значит, его можно было бы выполнить параллельно с чем-нибудь другим. Разумеется, на практике речь не идёт о чём-то столь тривиальном, как `1+2`, но сам факт наличия невыполненных вычислений, тем не менее, оказывается важным.

В этот момент мы говорим, что `x` *не вычислено*. Обычно в Haskell нельзя определить, вычислено ли `x`, но, к счастью, отладчик GHCi предоставляет несколько команд, с помощью которых можно проинспектировать структуру выражений на Haskell, не разрушая её. Будем использовать эти команды для иллюстрации происходящего. Команда `:sprint` выводит значение выражения, не пытаясь его предварительно вычислить:

```
Prelude> :sprint x
x = _
```

Специальный символ `_` означает «не вычислено». Другой, возможно, известный вам в этом контексте термин — «задумка» (*thunk*), который соответствует объекту в памяти, представляющему невыполненное вычисление `1+2`. Задумка в этом случае выглядит примерно так, как показано на рис. 2.1. Здесь `x` — это указатель на объект в памяти, представляющий функцию `+`, применяемую к двум целым `1` и `2`.

Задумка, представляющая `x`, будет вычислена, как только потребуется соответствующее значение. Проще всего заставить что-либо вычислиться — вывести значение, поэтому просто введём в командную строку интерпретатора `x`:

```
Prelude> x
3
```

Если теперь проинспектировать значение `x` с помощью `:sprint`, то станет ясно, что оно уже вычислено:

```
Prelude> :sprint x
x = 3
```

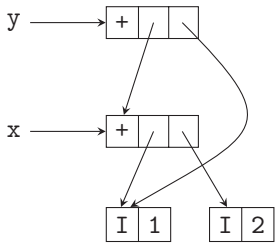

Рис. 2.2: одна задумка ссылается на другую

В терминах объектов в памяти можно говорить, что задумка, представлявшая `1+2`, перезаписана на упакованное (*boxed*) целое `3`[2]. Поэтому все последующие запросы значения `x` будут удовлетворяться немедленно, так работают ленивые вычисления.

Это был слишком простой пример. Попробуем нечто более сложное.

```
Prelude> let x = 1 + 2 :: Int
Prelude> let y = x + 1
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
```

Снова `x` связано с `1+2`, но теперь вдобавок `y` связано с `x+1`, а `:sprint`, как и ожидалось, показывает, что они оба не вычислены. В памяти возникает структура, показанная на рис. 2.2. К сожалению, нет прямого способа проинспектировать эту структуру в целом, поэтому вам придётся просто поверить мне на слово.

Теперь, чтобы вычислить значение `y`, требуется значение `x`: `y` зависит от `x`. Поэтому вычисление `y` приведёт к вычислению `x`. На этот раз воспользуемся другим способом заставить значение вычислиться — встроенной в Haskell функцией `seq`.

```
Prelude> seq y ()
()
```

[2] Строго говоря, она перезаписана на косвенную ссылку на значение, но детали здесь несущественны. Заинтересованные читатели могут обратиться либо к вики-страницам компилятора GHC и почитать там документацию по его реализации, либо к многочисленным статьям, посвящённым его проектированию.

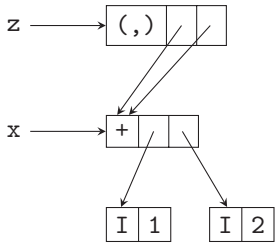

Рис. 2.3: пара, оба компонента которой ссылаются на одну и ту же задумку

Функция `seq` вычисляет свой первый параметр, в данном случае `y`, а затем возвращает свой второй параметр, здесь `()`. Проверим теперь значения переменных `x` и `y`:

```
Prelude> :sprint x
x = 3
Prelude> :sprint y
y = 4
```

Обе переменные, как и ожидалось, вычислились. Таким образом, пока мы пришли к следующим принципам:

- Определение выражения приводит к формированию в памяти задумки, представляющей это выражение.
- Задумка остаётся невычисленной до тех пор, пока не потребуется её значение. Будучи однажды вычисленной, задумка заменяется на своё значение.

Посмотрим, что происходит, когда добавляется структура данных:

```
Prelude> let x = 1 + 2 :: Int
Prelude> let z = (x,x)
```

Это определение связывает `z` с парой `(x,x)`. Команда `:sprint` показывает нечто интересное:

```
Prelude> :sprint z
z = (_,_)
```

Соответствующая структура показана на рис. 2.3.

Собственно переменная `z` ссылается на пару `(x,x)`, но оба компонента пары ссылаются на невычисленную задумку для `x`. Значит, можно строить структуры данных с невычисленными компонентами.

Давайте превратим `z` в задумку:

```
Prelude> import Data.Tuple
Prelude Data.Tuple> let z = swap (x,x+1)
```

Функция `swap` определена как `swap (a,b) = (b,a)`. Теперь `z` не вычислена:

```
Prelude Data.Tuple> :sprint z
z = _
```

Смысл этих действий в том, что теперь можно посмотреть, что происходит с переменной `z` при попытке вычислить её посредством `seq`:

```
Prelude Data.Tuple> seq z ()
()
Prelude Data.Tuple> :sprint z
z = (_,_)
```

Применение `seq` к `z` привело к тому, что `z` вычислилась как пара, но *её компоненты по-прежнему остались невычисленными*. Функция `seq` вычисляет свой параметр только до первого конструктора, оставляя более глубокие компоненты структуры невычисленными. Для обозначения такого поведения имеется специальный технический термин: говорят, что `seq` вычисляет свой параметр до *слабой головной нормальной формы* (*weak head normal form*, WHNF). Причины такой терминологии во многом исторические, поэтому не будем особенно о них задумываться. Термин *нормальная форма*, в свою очередь, означает «полностью вычислено». Мы научимся вычислять значения до нормальной формы в разделе «Модуль DeepSeq» на с. 49.

Понятие слабой головной нормальной формы возникнет в следующих двух главах несколько раз, поэтому стоит потратить время на его освоение, а заодно понять, как именно выполняются вычисления в программах на Haskell. Лучший способ для того и другого — поиграть в GHCi с выражениями и командой `:sprint`.

Закончим предыдущий пример, вычислив `x`:

```
Prelude Data.Tuple> seq x ()
()
```

Что мы теперь увидим, попросив вывести значение `z`?

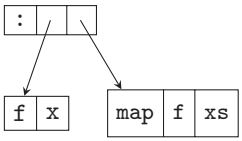

Рис. 2.4: задумка, созданная функцией `map`

```
Prelude Data.Tuple> :sprint z
z = (_,3)
```

Вспомним, что переменная `z` определялась как `swap (x,x+1)`, то есть `(x+1,x)`, а мы только что вычислили `x`, поэтому второй компонент пары `z` оказался вычисленным и равным 3.

Наконец, посмотрим на пример со списками и функциями на них. Вам наверняка известно определение функции `map`, приведём его здесь для справки.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Функция `map` строит ленивую структуру данных. Возможно, станет яснее, если мы перепишем определение `map` так, чтобы задумки стали явными:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = let
                   x'  = f x
                   xs' = map f xs
               in
                   x' : xs'
```

Поведение этого определения в точности такое же, как и у его исходного варианта, но теперь чётко видно, что и голова, и хвост списка, возвращаемого `map`, являются задумками `f x` и `map f xs` соответственно. Получается, что `map` строит в памяти структуру, показанную на рис. 2.4.

Определим с помощью `map` простую списочную структуру:

```
Prelude> let xs = map (+1) [1..10] :: [Int]
```

Пока ничего не вычислено:

```
Prelude> :sprint xs
xs = _
```

Вычислим теперь список до слабой головной нормальной формы:

```
Prelude> seq xs ()
()
Prelude> :sprint xs
xs = _ : _
```

Получаем список с как минимум одним элементом, но это всё, что мы к данному моменту о нём знаем. Теперь применим к списку функцию `length`:

```
Prelude> length xs
10
```

Функция `length` определяется следующим образом:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Обратите внимание, что `length` игнорирует голову списка, организуя рекурсивный обход хвоста `xs`. Поэтому в результате применения `length` к списку она обойдёт весь список, вычисляя позиции в списке, но не значения его элементов. Эффект обхода легко увидеть с помощью `:sprint`:

```
Prelude> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]
```

Интерпретатор GHCi замечает, что все позиции в списке вычислены, поэтому он, отказавшись от инфиксного `:`, переходит к обозначению списка с помощью квадратных скобок.

Хотя мы только что вычислили, фактически, скелет списка, он всё ещё не в нормальной форме (правда, он остаётся в слабой головной нормальной форме). Заставить его вычислиться целиком можно, применив функцию, которой требуются значения элементов, например `sum`:

```
Prelude> sum xs
65
Prelude> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Мы лишь чуть-чуть вскрыли поверхность довольно тонкой и сложной темы. К счастью, большую часть времени при написании кода на Haskell

о порядке вычисления значения беспокоиться не приходится. В самом деле, само определение языка Haskell старается обходить любые указания на точный порядок вычислений, в результате реализация получает возможность выбора собственной стратегии вычислений в тех пределах, в которых программа выдаёт правильный результат. Именно об этом мы как программисты должны обычно заботиться. Однако при написании параллельного кода оказывается важным понимать, когда именно выполняются вычисления, только тогда их можно будет распараллелить.

У использования ленивых вычислений в параллельном коде есть альтернатива — можно явно задавать потоки данных, этот подход реализуется монадой `Par` в четвёртой главе. При этом можно избежать некоторых довольно тонких сложностей, касающихся ленивых вычислений, правда, за счёт большей многословности. Тем не менее полезно изучить оба подхода, поскольку встречаются ситуации, когда один из них оказывается более естественным или более эффективным, чем другой.

Монада Eval, rpar и rseq

Введём теперь самый простой функционал для описания параллелизма, предоставляемый модулем `Control.Parallel.Strategies`:

```
data Eval a

instance Monad Eval

runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

Параллелизм реализуется с помощью монады `Eval`, которая определяет две операции, `rpar` и `rseq`. Комбинатор `rpar` описывает параллельное вычисление, он как бы утверждает: «Мой параметр может выполняться параллельно». В свою очередь, `rseq` используется для явного задания последовательного вычисления: «Запусти вычисление моего параметра и дождись результата». В каждом из случаев вычисление осуществляется до слабой головной нормальной формы. Стоит также иметь в виду, что параметром `rpar` должна быть задумка — невычисленное выражение. Если параметр уже вычислен, то ничего не произойдёт, поскольку нет работы, которую можно было бы выполнить параллельно.

Операция `runEval` монады `Eval` выполняет заданное вычисление и возвращает его результат. Заметьте, что `runEval` — чистая функция, здесь

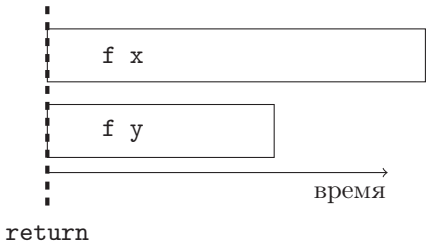

Рис. 2.5: временная шкала для варианта `rpar/rpar`

нет необходимости в монаде `IO`.

Разберёмся с действием `rpar` и `rseq`. Предположим, что у нас есть функция `f` и два параметра `x` и `y`. Мы хотим параллельно вычислить `f x` и `f y`. Допустим, `f x` вычисляется дольше, чем `f y`. Мы посмотрим на несколько разных способов программирования требуемого поведения и изучим различия между ними. Во-первых, воспользуемся для вычисления обоих значений комбинатором `rpar` (схема выполнения этого фрагмента программы приведена на рис. 2.5):

Пример 2.1: rpar/rpar

```
runEval $ do
   a <- rpar (f x)
   b <- rpar (f y)
   return (a,b)
```

Видно, что `f x` и `f y` начинают выполняться параллельно, а `return` срабатывает сразу, не дожидаясь, когда закончится вычисление `f x` или `f y`. Остаток программы будет выполняться параллельно с вычислением `f x` и `f y`.

Попробуем другой вариант, заменив второй `rpar` на `rseq` (схема выполнения показана на рис. 2.6):

Пример 2.2: rpar/rseq

```
runEval $ do
   a <- rpar (f x)
   b <- rseq (f y)
   return (a,b)
```

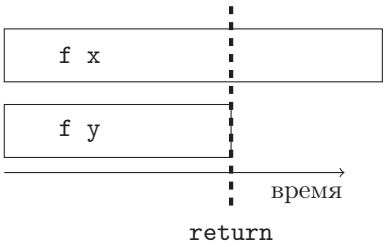

Рис. 2.6: временная шкала для варианта `rpar/rseq`

В этом варианте `f x` и `f y` по-прежнему вычисляются параллельно, но теперь `return` не срабатывает вплоть до завершения вычисления `f y`. Это результат действия комбинатора `rseq`, который ожидает завершения вычисления значения своего параметра.

Если добавить вызов дополнительного `rseq`, ожидающего `f x`, то окажется, что мы ждём завершения обоих вычислений:

Пример 2.3: rpar/rseq/rseq

```
runEval $ do
   a <- rpar (f x)
   b <- rseq (f y)
   rseq a
   return (a,b)
```

Обратите внимание, что `rseq` здесь применяется к `a`, то есть к результату вычисления первого `rpar`. В итоге получаем порядок, представленный на рис. 2.7. Этот код ожидает завершения вычисления `f x` и `f y`, а затем делает вызов `return`.

Каким из этих способов стоит пользоваться?

- Вариант `rpar/rseq` вряд ли будет полезен, потому что программист довольно редко заранее знает, какое из вычислений займёт больше времени. Немного смысла в том, чтобы дожидаться завершения одного произвольного вычисления из двух.

- Выбор между `rpar/rpar` и `rpar/rseq/rseq` зависит от обстоятельств. Если в дальнейшем в программе параллелизм будет, и последующие вычисления не зависят непосредственно от результатов этого, то имеет смысл использовать `rpar/rpar`, который отдаёт управление сразу

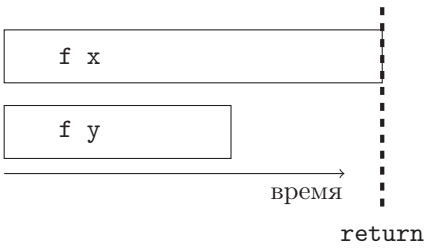

Рис. 2.7: временная шкала для варианта `rpar/rseq/rseq`

после запуска вычислений. С другой стороны, если больше параллелизма не предвидится или же результаты одной из операций требуются для продолжения работы, то `rpar/rseq/rseq` позволяет явно об этом заявить.

Вот ещё один заключительный вариант:

Пример 2.4: rpar/rpar/rseq/rseq

```
runEval $ do
   a <- rpar (f x)
   b <- rpar (f y)
   rseq a
   rseq b
   return (a,b)
```

Поведение здесь совпадает с вариантом `rpar/rseq/rseq`, поскольку ожидается завершение каждого из вычислений. Хотя этот пример и самый длинный, зато в нём больше симметрии, чем во всех остальных, по этой причине иногда его стоит предпочесть.

Для самостоятельных экспериментов с приведёнными способами описания параллельных и последовательных вычислений можно воспользоваться программой *rpar.hs*, в которой для симуляции затратных вычислений применяются числа Фибоначчи, вычисляемые параллельно. Параллелизм в GHC необходимо включать параметром командной строки `-threaded`. Скомпилируйте программу следующим образом:

```
$ ghc -O2 rpar.hs -threaded
```

Чтобы посмотреть на вариант `rpar/rpar`, программу следует запускать так:

```
$ ./rpar 1 +RTS -N2
time: 0.00s
(24157817,14930352)
time: 0.83s
```

Флаг `+RTS -N2` указывает GHC, что для исполнения программы необходимо использовать два ядра (считаем, что у вас как минимум двухъядерный процессор). Первая отметка времени печатается при выходе из `rpar/rseq`-фрагмента, а вторая — в момент завершения вычислений. Здесь видно, что `return` срабатывает немедленно. В варианте `rpar/rseq` это происходит после завершения второго (более короткого) вычисления:

```
$ ./rpar 2 +RTS -N2
time: 0.50s
(24157817,14930352)
time: 0.82s
```

Наконец, в варианте `rpar/rseq/rseq` выполнение `return` происходит в самом конце:

```
$ ./rpar 3 +RTS -N2
time: 0.82s
(24157817,14930352)
time: 0.82s
```

Пример: распараллеливание решателя судоку

В этом разделе мы будем разбирать большой пример, на котором рассмотрим способы добавления параллелизма в программу, выполняющую одно и то же вычисление над большим набором входных данных. Вычислением будет реализация решателя судоку. Наш решатель довольно быстр, он способен решить все 49 000 известных заданий с 17 подсказками примерно за две минуты.

Целью будет распараллеливание решения разных заданий. Не важно, как именно решается одно задание, мы будем считать решателя чёрным ящиком. Для нас это просто пример затратного вычисления, которое можно запускать на большом наборе данных, а именно на заданиях судоку.

Воспользуемся модулем `Sudoku` и функцией `solve` из него с типом:

```
solve :: String -> Maybe Grid
```

Значение типа `String` соответствует одному заданию, это линеаризованное представление поля размером 9 на 9, каждый квадрат которого либо пуст, что соответствует символу `'.'`, либо содержит цифру от 1 до 9.

Функция `solve` возвращает значение типа `Maybe Grid`, это либо `Nothing`, если задача не имеет решения, либо `Just g`, где `g` типа `Grid`, если решение найдено. В этом примере нас не интересует само решение, важно лишь, существует ли оно вообще.

Начнём с обычного последовательного кода, решающего набор судоку, считываемый из файла:

sudoku1.hs

```
import Sudoku
import Control.Exception
import System.Environment
import Data.Maybe

main :: IO ()
main = do
  [f] <- getArgs                                -- (1)
  file <- readFile f                            -- (2)

  let  puzzles   = lines file                   -- (3)
       solutions = map solve puzzles            -- (4)

  print (length (filter isJust solutions))      -- (5)
```

Эта короткая программа работает следующим образом:

(1) Получаем параметры командной строки, ожидая только один — имя файла с данными.

(2) Читаем содержимое заданного файла.

(3) Делим файл на строки, одна строка — одно задание.

(4) Решаем все задания, пропуская функцию `solve` по списку строк.

(5) Подсчитываем число заданий, имеющих решение, удаляя сначала все `Nothing`, а затем считая и печатая длину полученного списка. Хотя самими решениями мы здесь не интересуемся, `filter isJust` необходима: без неё программа не стала бы вычислять элементы списка, и решатель ничего бы не делал (вспомните пример с `length` в конце раздела «Ленивые вычисления и слабая головная нормальная форма» на с. 26).

Давайте проверим, как справится эта программа с конкретным набором заданий. Для начала её следует откомпилировать: