

Лекция 13: многопоточность

Функциональное программирование на Haskell

Алексей Романов

6 марта 2023 г.

МИЭТ

Два вида многопоточности: конкурентность и параллелизм

- Все программы, которые мы писали до сих пор, были однопоточными.
- У современных компьютеров много ядер, мы задействуем только одно.
- Чтобы использовать их, нужна многопоточность.

Два вида многопоточности: конкурентность и параллелизм

- Все программы, которые мы писали до сих пор, были однопоточными.
- У современных компьютеров много ядер, мы задействуем только одно.
- Чтобы использовать их, нужна многопоточность.
- Параллелизм: физическая многопоточность. Много ядер (или других ресурсов) используются для ускорения выполнения одной задачи.

Два вида многопоточности: конкурентность и параллелизм

- Все программы, которые мы писали до сих пор, были однопоточными.
- У современных компьютеров много ядер, мы задействуем только одно.
- Чтобы использовать их, нужна многопоточность.
- Параллелизм: физическая многопоточность. Много ядер (или других ресурсов) используются для ускорения выполнения одной задачи.
- Конкурентность: логическая многопоточность. Несколько задач, которые могут выполняться логически одновременно. А значит, потенциально и физически одновременно.

Детерминированность

- Модель программирования детерминирована, если результат программы зависит только от полученных данных (аргументов и ввода пользователя).

Детерминированность

- Модель программирования детерминирована, если результат программы зависит только от полученных данных (аргументов и ввода пользователя).
- Недетерминирована, если результат зависит от того, в каком порядке произойдут события (или от сгенерированных случайных чисел и т.д.).

Детерминированность

- Модель программирования детерминирована, если результат программы зависит только от полученных данных (аргументов и ввода пользователя).
- Недетерминирована, если результат зависит от того, в каком порядке произойдут события (или от сгенерированных случайных чисел и т.д.).
- Обычно параллельные программы детерминированы, конкурентные нет.
- Недетерминированность усложняет тестирование, отладку и понимание программ, но часто без неё не обойтись.
- Тестирование свойств это пример пользы от недетерминированности.

Параллелизм: монада `Eval`

- Модуль `Concurrent.Parallel.Strategies` в пакете `parallel`.
- Тип `Eval` а — монада, где в `mx >=> f` значение `mx` будет вычислено до СЗНФ перед запуском `f`.

Параллелизм: монада `Eval`

- Модуль `Concurrent.Parallel.Strategies` в пакете `parallel`.
- Тип `Eval a` — монада, где в `mx >= f` значение `mx` будет вычислено до СЗНФ перед запуском `f`.
- Есть функции
`runEval :: Eval a -> a`
`rpar :: a -> Eval a`
`rseq :: a -> Eval a`
`parEval :: Eval a -> Eval a`
- `rpar x`: запустить вычисление `x` и сразу вернуться.
- `rseq x`: вернуться после вычисления `x`.
- `parEval x`: запустить вычисление, описанное в `x`, и сразу вернуться.

Примеры Eval

- Запуск двух вычислений параллельно:

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

Примеры Eval

- Запуск двух вычислений параллельно:

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

- Запуск двух вычислений параллельно и возврат, когда оба закончатся:

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

Фибоначчи в Eval

- `parFib1 :: Integer -> Integer`
`parFib1 0 = 1`
`parFib1 1 = 1`
`parFib1 n = runEval $ do`

Фибоначчи в Eval

- ```
parFib1 :: Integer -> Integer
parFib1 0 = 1
parFib1 1 = 1
parFib1 n = runEval $ do
 f1 <- rpar (parFib1 (n - 1))
 f2 <- rseq (parFib1 (n - 2))
 -- rseq, чтобы занять текущий поток, можно и rpar
 pure (f1 + f2)
```

## Фибоначчи в Eval

- `parFib1 :: Integer -> Integer`  
`parFib1 0 = 1`  
`parFib1 1 = 1`  
`parFib1 n = runEval $ do`  
    `f1 <- rpar (parFib1 (n - 1))`  
    `f2 <- rseq (parFib1 (n - 2))`  
    *-- rseq, чтобы занять текущий поток, можно и rpar*  
    `pure (f1 + f2)`
- `parFib2 :: Integer -> Eval Integer`  
`parFib2 0 = pure 1`  
`parFib2 1 = pure 1`  
`parFib2 n = do`

## Фибоначчи в Eval

- `parFib1 :: Integer -> Integer`  
`parFib1 0 = 1`  
`parFib1 1 = 1`  
`parFib1 n = runEval $ do`  
    `f1 <- rpar (parFib1 (n - 1))`  
    `f2 <- rseq (parFib1 (n - 2))`  
    *-- rseq, чтобы занять текущий поток, можно и rpar*  
    `pure (f1 + f2)`
- `parFib2 :: Integer -> Eval Integer`  
`parFib2 0 = pure 1`  
`parFib2 1 = pure 1`  
`parFib2 n = do`  
    `f1 <- parEval (parFib2 (n - 1))`  
    `f2 <- parFib2 (n - 2)`  
    `pure (f1 + f2)`

## Фибоначчи в Eval

- `parFib1 :: Integer -> Integer`  
`parFib1 0 = 1`  
`parFib1 1 = 1`  
`parFib1 n = runEval $ do`  
    `f1 <- rpar (parFib1 (n - 1))`  
    `f2 <- rseq (parFib1 (n - 2))`  
    *-- rseq, чтобы занять текущий поток, можно и rpar*  
    `pure (f1 + f2)`
- `parFib2 :: Integer -> Eval Integer`  
`parFib2 0 = pure 1`  
`parFib2 1 = pure 1`  
`parFib2 n = do`  
    `f1 <- parEval (parFib2 (n - 1))`  
    `f2 <- parFib2 (n - 2)`  
    `pure (f1 + f2)`
- Здесь для простоты кода много повторных вычислений!



# Стратегии вычислений

- `type Strategy a = a -> Eval a`.
- `rpar` и `rseq` — стратегии.
- `r0` — стратегия, которая ничего не вычисляет.
- Стратегия принимает на вход `thunk`, вычисляет его части с помощью `r0`, `rpar` и `rseq`.
- Например, обобщая пример с прошлого слайда  
`rparTuple2 :: Strategy (a, b)`  
`rparTuple2 (a, b) = do`

# Стратегии вычислений

- `type Strategy a = a -> Eval a.`
- `rpar` и `rseq` — стратегии.
- `r0` — стратегия, которая ничего не вычисляет.
- Стратегия принимает на вход `thunk`, вычисляет его части с помощью `r0`, `rpar` и `rseq`.
- Например, обобщая пример с прошлого слайда  
`rparTuple2 :: Strategy (a, b)`  
`rparTuple2 (a, b) = do`  
    `a' <- rpar a`  
    `b' <- rpar b`  
    `return (a', b')`

# Стратегии вычислений

- `type Strategy a = a -> Eval a`.
- `rpar` и `rseq` — стратегии.
- `r0` — стратегия, которая ничего не вычисляет.
- Стратегия принимает на вход `thunk`, вычисляет его части с помощью `r0`, `rpar` и `rseq`.
- Например, обобщая пример с прошлого слайда

```
rparTuple2 :: Strategy (a, b)
```

```
rparTuple2 (a, b) = do
```

```
 a' <- rpar a
```

```
 b' <- rpar b
```

```
 return (a', b')
```

```
(или rparTuple2 (a, b) = liftA2 (,) (rpar a)
 (rpar b))
```

- `withStrategy :: Strategy a -> a -> a`  
`withStrategy s x = runEval (s x)`

# Комбинаторы стратегий

- Обобщим `rparTuple2` дальше, чтобы она принимала стратегии на вход:

```
evalTuple2 :: Strategy a -> Strategy b ->
 Strategy (a, b)
evalTuple2 strat_a strat_b (a, b) =
```

## Комбинаторы стратегий

- Обобщим `rparTuple2` дальше, чтобы она принимала стратегии на вход:

```
evalTuple2 :: Strategy a -> Strategy b ->
 Strategy (a, b)
evalTuple2 strat_a strat_b (a, b) =
 liftA2 (,) (strat_a a) (strat_b b)
```

# Комбинаторы стратегий

- Обобщим `rparTuple2` дальше, чтобы она принимала стратегии на вход:  
`evalTuple2 :: Strategy a -> Strategy b -> Strategy (a, b)`  
`evalTuple2 strat_a strat_b (a, b) = liftA2 (,) (strat_a a) (strat_b b)`
- `rparWith :: Strategy a -> Strategy a` запускает стратегию параллельно.
- Например, `rparWith rpar` и `rparWith rseq` эквивалентны `rpar`.

# Комбинаторы стратегий

- Обобщим `rparTuple2` дальше, чтобы она принимала стратегии на вход:

```
evalTuple2 :: Strategy a -> Strategy b ->
 Strategy (a, b)
```

```
evalTuple2 strat_a strat_b (a, b) =
 liftA2 (,) (strat_a a) (strat_b b)
```

- `rparWith :: Strategy a -> Strategy a` запускает стратегию параллельно.
- Например, `rparWith rpar` и `rparWith rseq` эквивалентны `rpar`.

- `parTuple2 :: Strategy a -> Strategy b ->  
 Strategy (a, b)`

```
parTuple2 strat_a strat_b =
 evalTuple2 (rparWith strat_a) (rparWith strat_b)
```

## Комбинаторы стратегий для списков

- С помощью стратегий достаточно легко сделать параллельные `map/filter`/и т.д.
- Например, используя `parList :: Strategy a -> Strategy [a]`:  
`parallelMap :: (a -> b) -> [a] -> [b]`  
`parallelMap f xs =`  
    `withStrategy (parList rseq) (map f xs)`



## Комбинаторы стратегий для списков

- С помощью стратегий достаточно легко сделать параллельные `map/filter`/и т.д.
- Например, используя `parList :: Strategy a -> Strategy [a]`:  
`parallelMap :: (a -> b) -> [a] -> [b]`  
`parallelMap f xs =`  
    `withStrategy (parList rseq) (map f xs)`
- Это скорее всего создаст слишком много «искр», лучше использовать
  - `parListChunk`: разбивает список на части одинаковой длины и работает параллельно с каждой частью
  - `parBuffer`: начинает работать параллельно с первыми элементами списка, добавляет следующие при использовании результатов

## Монада **Par**: параллелизм без ленивости

- Модуль **Control.Monad.Par** в пакете **monad-par**.
- Тип **Par a** — тоже монада, описывающая процесс вычисления значения типа **a** с параллельными частями.
- **fork :: Par () -> Par ()** создаёт новый поток.
- **runPar :: Par a -> a** производит вычисление и возвращает его результат.

## Монада **Par**: параллелизм без ленивости

- Модуль **Control.Monad.Par** в пакете **monad-par**.
- Тип **Par a** — тоже монада, описывающая процесс вычисления значения типа **a** с параллельными частями.
- **fork :: Par () -> Par ()** создаёт новый поток.
- **runPar :: Par a -> a** производит вычисление и возвращает его результат.
- Потоки общаются между собой с помощью **IVar**, которые можно записать только 1 раз.
- В результате не должно быть **IVar** или ссылок на них!
- При соблюдении этого условия вычисления в **Par** детерминированы.

- Интерфейс `IVar`:

```
new :: Par (IVar a)
newFull :: NFData a => a -> Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

- Значения, положенные в `IVar`, полностью вычисляются (есть ещё `newFull_` и `put_`).
- Переменной можно дать значение только 1 раз, иначе исключение.
- При вызове `get` для переменной, ещё не имеющей значения, начинается ожидание.

## Примеры Par

- `spawn :: NFData a => Par a -> Par (IVar a)`:  
аналог `fork`, но возвращающий переменную, в которой будет сохранено вычисленное значение  
`spawn par = do`  
    `var <- new`  
    `fork $ do (x <- par; put var x)`  
    `return var`
- Параллельный `map`:  
`parallelMap :: NFData b => (a -> b) -> [a] -> [b]`  
`parallelMap f xs = runPar $ do`  
    `ivars <- mapM (spawn (\x -> pure (f x)) xs`  
    `mapM get ivars`  
Как и в прошлый раз, создаёт «поток» для каждого элемента, для простоты кода.

## Фибоначчи в Par

```
parFib :: Integer -> Par Integer
parFib 0 = pure 1
parFib 1 = pure 1
parFib n = do
```

```
parFib :: Integer -> Par Integer
parFib 0 = pure 1
parFib 1 = pure 1
parFib n = do
 f1var <- spawn (parFib (n - 1))
 f2 <- parFib (n - 2)
 f1 <- get f1var
 pure (f1 + f2)
```

# Конкурентность

- Перейдём от параллелизма к конкурентности.
- Модуль `Control.Concurrent` в пакете `base` и `C.C.MVar/C.C.Chan/C.C.QSem[N]`.
- `ThreadId` — ссылка на поток.
- `forkIO :: IO () -> IO ThreadId` создаёт новый поток, который произведёт данные действия и выйдет, возвращает ссылку на созданный поток.



# Конкурентность

- Перейдём от параллелизма к конкурентности.
- Модуль `Control.Concurrent` в пакете `base` и `C.C.MVar/C.C.Chan/C.C.QSem[N]`.
- `ThreadId` — ссылка на поток.
- `forkIO :: IO () -> IO ThreadId` создаёт новый поток, который произведёт данные действия и выйдет, возвращает ссылку на созданный поток.
- Это так называемый «зелёный поток», то есть не поток операционной системы, а похожий по поведению объект виртуальной машины Haskell.

- Потоки могут взаимодействовать через **MVar**, которые очень похожи на **IVar**, но записанное значение можно убрать или изменить.
- Переменная в любой момент или пустая или полная (содержит значение).

```
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
putMVar :: MVar a -> a -> IO ()
readMVar :: MVar a -> IO a
takeMVar :: MVar a -> IO a
isEmptyMVar :: MVar a -> IO Bool
```

- Если **putMVar** вызвана на полной переменной, она ждёт, пока та не опустеет.
- **takeMVar** и **readMVar** отличаются тем, что первая убирает прочитанное значение. На пустой переменной обе ждут.

## Фибоначчи с **MVar**

- Реализация вычисления чисел Фибоначчи с помощью **MVar** не сильно отличается от **Par** (только аналога **spawn** в стандартной библиотеке нет, смотрите пакет **async**):

```
parFib :: Integer -> IO Integer
```

```
parFib 0 = pure 1
```

```
parFib 1 = pure 1
```

```
parFib n = do
```

## Фибоначчи с MVar

- Реализация вычисления чисел Фибоначчи с помощью MVar не сильно отличается от Par (только аналога spawn в стандартной библиотеке нет, смотрите пакет async):

```
parFib :: Integer -> IO Integer
parFib 0 = pure 1
parFib 1 = pure 1
parFib n = do
 flvar <- newEmptyMVar
 forkIO $ do
 f1' <- parFib (n - 1)
 putMVar flvar f1'
 f2 <- parFib (n - 2)
 f1 <- readMVar flvar
 pure (f1 + f2)
```

# Правильный Фибоначчи

- Уже упоминалось, что приведённые программы для вычисления чисел Фибоначчи имеют серьёзный недостаток.
- Например, `parFib 4` вызовет параллельно `parFib 3` и `parFib 2`, а `parFib 3` снова вызовет `parFib 2`.
- Чтобы этого избежать, нужно добавить вспомогательный аргумент для хранения уже полученных результатов.
- Его тип может быть:
  - Для `Eval: Map Integer Integer`.
  - Для `Par: Map Integer (IVar Integer)`.
  - Для `IO: MVar (Map Integer Integer)`, хотя `Map Integer (MVar Integer)` тоже подойдёт.

- **MVar** можно рассматривать как каналы, хранящие не более одного значения.
- **Chan** это канал для произвольного количества значений.

```
newChan :: IO (Chan a)
```

```
writeChan :: Chan a -> a -> IO ()
```

```
readChan :: Chan a -> IO a
```

- **writeChan** сохраняет значение в конец очереди.
- **readChan** читает из её начала и ждёт, если она пуста.

# Транзакции

- Понятие транзакций может быть знакомо по базам данных.
- Это наборы операций, которые все вместе либо выполняются успешно, либо откатываются.
- В Haskell это позволяет делать монада **STM** с операциями над **TVar**.
- Типы гарантируют, что у действия в монаде **STM** не может быть побочных эффектов, кроме изменения значений **TVar**.
- Обычно проще написать правильный код, который работает с несколькими **TVar**, чем с **MVar**, так как можно не беспокоиться о видимости промежуточных состояний.

## Дополнительное чтение

- [Parallel and Concurrent Programming in Haskell](#) (книга, 2013, доступна бесплатно)
- [A Tutorial on Parallel and Concurrent Programming in Haskell](#)
- [Erlang](#) Функциональный язык, построенный на модели акторов (взаимодействующих только через отправку сообщений друг другу вместо общих переменных).
- [Ответ на Stack Overflow про разницу между Eval и Par](#) (ещё один в конце главы 4 первой ссылки).
- [Streamly: Beautiful Streaming, Concurrent and Reactive Composition](#)
- [Ivish](#) Пакет, обобщающий [Par](#) на основе теории решёток. К сожалению, мёртв с 2014.