CS314. Функциональное программирование

Лекция 20. Параллельное программирование (2). Исключения в Haskell

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

10 декабря 2016 г.

Содержание

Содержание

Содержание

Модуль Control.Parallel.Strategies

```
data Eval a
instance Monad Eval

type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
r0 :: Strategy a
rseq :: Strategy a
rdeepseq :: NFData a => Strategy a
rpar :: a -> Eval a
rparWith :: Strategy a -> Strategy a
evalList :: Strategy a -> Strategy [a]
parList :: Strategy a -> Strategy [a]
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
```

Содержание

Интерфейс монады Par (Control.Monad.Par)

```
newtype Par a
instance Applicative Par
instance Monad Par

runPar :: Par a -> a

fork :: Par () -> Par ()
```

```
data IVar a

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

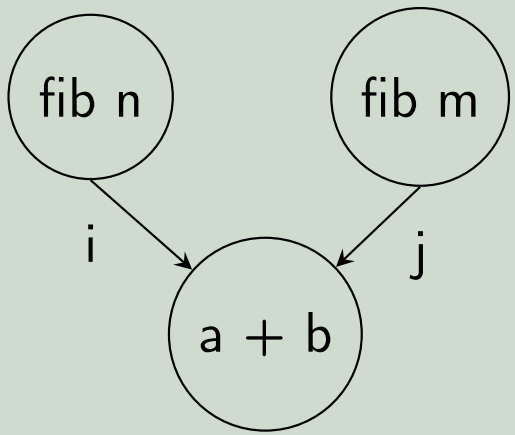
Записать IVar можно только один раз!

Пример: параллельное вычисление чисел Фибоначчи

```
fib :: Int -> Integer

fibSum :: Int -> Int -> Integer
fibSum n m = runPar $ do
      i <- new
      j <- new
      fork (put i (fib n))
      fork (put j (fib m))
      a <- get i
      b <- get j
      return (a+b)
```

Граф потоков данных



- Текст программы описывает зависимости между данными.
- Библиотека определяет, что может быть вычислено параллельно.

Параллельное вычисление элементов списка

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

Монада Par: заключение

- Описываем вычисление в виде потоков данных.
- Библиотека выполняет все вычисления по возможности параллельно.

Содержание

Массивы библиотеки Repa (Data.Array.Repa)

Тип для массива

```
data Array r sh e
```

- r — представление в памяти
- sh — форма
- e — тип элементов (Double, Int, Word8)

Формы

```
data Z = Z
data tail :. head = tail :. head

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
```

Формирование массива и доступ к элементам

```
fromListUnboxed :: (Shape sh, Unbox a) =>
                                sh -> [a] -> Array U sh a
```

```
ghci> fromListUnboxed (Z :. 10) [1..10] :: Array U DIM1 Int
AUnboxed (Z :. 10) (fromList [1,2,3,4,5,6,7,8,9,10])
```

```
ghci> let arr = fromListUnboxed (Z :. 3 :. 5) [1..15]
                                        :: Array U DIM2 Int
ghci> arr
AUnboxed ((Z:.3):.5) (fromList [1,2,3,4,5,6,7,8,9,10,...,15])
```

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

```
ghci> arr ! (Z:.2:.1)
12
```

Операции над массивами

```
Repa.map :: (Shape sh, Source r a)
         => (a -> b) -> Array r sh a -> Array D sh b
```

- D — представление для отложенного массива (он не существует в памяти).
- Идея: мы можем вызвать map несколько раз, описывая таким образом сложные вычисления, но в действительности они выполняться не будут.

```
Repa.map (+1) (Repa.map (^2) a)
```

Выполнение операций над массивами

```
ghci> let a = fromListUnboxed (Z :. 10) [1..10]
                                        :: Array U DIM1 Int
ghci> computeS $ Repa.map (+1) $ Repa.map (^2) a
                                        :: Array U DIM1 Int
AUnboxed (Z :. 10) (fromList [2,5,10,17,26,37,50,65,82,101])
```

```
computeS :: (Load r1 sh e, Target r2 e) =>
                    Array r1 sh e -> Array r2 sh e
```

```
computeP :: (Monad m, Source r2 e, Target r2 e, Load r1 sh e)
         => Array r1 sh e -> m (Array r2 sh e)
```

- Распараллеливание с использованием всех доступных ресурсов!

Свёрточные операции

```
sumAllS
  :: (Num a, Shape sh, Source r a, Unbox a, Elt a)
  => Array r sh a
  -> a

foldS :: (Shape sh, Source r a, Elt a, Unbox a)
      => (a -> a -> a)
      -> a
      -> Array r (sh :. Int) a -> Array U sh a

foldP :: (Shape sh, Source r a, Elt a, Unbox a, Monad m)
      => (a -> a -> a)
      -> a
      -> Array r (sh :. Int) a -> m (Array U sh a)
```

Библиотека Repa: заключение

- Описываем (декларативно!) сложные вычисления над массивами.
- Запускаем вычисления, при этом:
  - циклы по возможности сливаются в один (fusion);
  - используются все доступные вычислительные ресурсы.
- В библиотеке поддерживаются много других видов вычислений помимо map/fold.

Содержание

Запуск простого вычисления на GPU

```
ghci> import Data.Array.Accelerate as A
ghci> import Data.Array.Accelerate.Interpreter as I
ghci> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
ghci> run $ A.map (+1) (use arr)
Array (Z :. 3 :. 5) [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

```
use :: Arrays arrays => arrays -> Acc arrays
run :: Arrays a => Acc a -> a
A.map :: (Shape ix, Elt a, Elt b)
      => (Exp a -> Exp b)
      -> Acc (Array ix a)
      -> Acc (Array ix b)
```

Содержание

Класс и типы исключений

```haskell
import Control.Exception
import Data.Typeable
```

```haskell
class (Typeable e, Show e) => Exception e where
  ...

data SomeException = forall e. Exception e => SomeException e
  deriving Typeable

data IOException
data ArithException
data BlockedIndefinitelyOnMVar
newtype ErrorCall
...
```

Генерация исключений

```
throw :: Exception e => e -> a

throwIO :: Exception e => e -> IO a
```

- Параметр и результат throw: исключение и любой тип
- Исключения в чистом коде и в монаде IO

Исключение ErrorCall

Тип ErrorCall

```
newtype ErrorCall = ErrorCall String
    deriving (Typeable)

instance Show ErrorCall

instance Exception ErrorCall
```

Функция error

```
error :: String -> a
error s = throw (ErrorCall s)
```

Перехват исключений

Перехват исключений возможен только в монаде IO!

```
catch ::
  Exception e =>
       IO a -> (e -> IO a) -> IO a
catch  action  handler = ...
```

- В обработчике необходимо указывать тип исключения.

```
  readFile f `catch`
     (\e -> do
        let err = show (e :: IOException)
        hPutStr stderr ("Warning: Couldn't open " ++
                        f ++ ": " ++ err)
        return "")
```

Собственное исключение и его перехват

```
data MyException = MyException deriving (Show, Typeable)
instance Exception MyException
```

```
ghci> throw MyException
*** Exception: MyException
ghci> throw MyException `catch` \e -> print e
  <interactive>:10:33:
    Ambiguous type variable `a0' in the constraints ...
ghci> throw MyException `catch`
                          \e -> print (e :: MyException)
MyException
ghci> throw (ErrorCall "oops") `catch`
                          \e -> print (e :: MyException)
*** Exception: oops
```

Перехват всех исключений

```
ghci> throw (ErrorCall "oops") `catch`
                              \e -> print (e :: SomeException)
oops
```

- Игнорирование ошибки
- Допустимо в двух случаях:
  - в процессе тестирования и отладки
  - при выполнении зачистки перед повторной генерацией исключения

Другие способы перехвата исключения

Функция try

```
try :: Exception e => IO a -> IO (Either e a)
```

```
ghci> try (readFile "nonexistent")
                    :: IO (Either IOException String)
Left nonexistent: openFile: does not exist
                            (No such file or directory)
```

Функция handle

```
handle :: Exception e => (e -> IO a) -> IO a -> IO a

handle (\e -> ...) $ do
...
```

Функция onException и повторная генерация исключения

```
onException :: IO a -> IO b -> IO a
onException io what
   = io `catch`
          \e -> do
             _ <- what
             throwIO (e :: SomeException)
```

Высокоуровневая обработка исключений

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c

finally :: IO a -> IO b -> IO a
```

- Параметры bracket: выделение ресурса, освобождение ресурса (зачистка), операция с ресурсом
- Параметры finally: операция и действия по зачистке

```
bracket (newTempFile "temp")
        (\file -> removeFile file)
        (\file -> ...)
```

Библиография

1. С. Марлоу. Параллельное и конкурентное программирование на языке Haskell. Текст на английском доступен онлайн: `http://chimera.labs.oreilly.com/books/1230000000929/index.html`