Несколько коротких тем

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

14 ноября 2019 г.

В этих слайдах

Хвостовая рекурсия

Очень наивный факториал в строгом языке

```
fac 0 = 1
fac n = n*fac (n-1)
```

Как же он выполнится ?

```
fac 5 = 5*(fac 4)
      = 5*(4*(fac 3))
      = 5*(4*(3*(fac 2)))
      = 5*(4*(3*(2*(fac 1))))
      = 5*(4*(3*(2*(1*(fac 0)))))
      = 5*(4*(3*(2*(1*1))))
      = 5*(4*(3*(2*1))) = 5*(4*(3*2)) = 5*(4*6) = 5*24 = 120
```

Данная реализация сильно расходует стек при вычислении

Канонический способ это исправить – переписать реализацию с использованием *хвостовой рекурсии*.

```haskell
facAux :: Int -> Int -> Int
facAux 0 r = r
facAux n r = facAux (n-1) (r*n)

fac n = facAux n 1
```

Необходимо, чтобы выражением-результатом функции было либо “простое” значение, либо рекурсивный вызов функции от аргументов, в которых не вызывается рекурсивно данная функция.

Тогда runtime сможет сэкономить на манипулировании аргументами на стеке и работать с производительностью как у цикла.

Лирическое отступление

Как лучше перемножать числа в списке? С помощью `foldr` ...

```
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs

sum1 = foldr (+) 0
```

... или с помощью `foldl`?

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x
                   in foldl f z' xs

sum2 = foldl (+) 0
```

Правильный ответ: с помощью

Лирическое отступление

Как лучше перемножать числа в списке? С помощью `foldr` ...

```
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs

sum1 = foldr (+) 0
```

... или с помощью `foldl`?

```
foldl f z []     = z
foldl f z (x:xs) = let z' = z `f` x
                   in foldl f z' xs

sum2 = foldl (+) 0
```

Правильный ответ: с помощью `foldl'`

Более сложный случай – 2 рекурсивных вызова

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Такая реализация фибоначчи выполняет рекурсию по древовидной структуре и тормозит.

К счастью, понятно как её переписать, чтобы получилась рекурсия по линейной структуре

```
fib (0), fib (1), fib (2), ... , fib (n-1), fib (n)
```

А именно

```
fibAux 0 result _    = result
fibAux 0 result prev = fibAux (n-1) (result + prev) result

fibTail 0 = 0
fibTail n = fibAux n 1 0
```

Здесь на нас сошло озарение, и мы смогли написать код.

Можно попробовать применить *кеширование*

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

```haskell
memoize :: (Int -> a) -> (Int -> a)
memoize f = (map f [0 ..] !!)
```

Надо только вызывать по-другому....

```haskell
fibCached = memoize fib -- будет запоминать результаты между вызовами
                        -- но не так хорошо, как хотелось бы
```

Эта штука не кеширует вызовы *внутри*

Вначале перепишем fib, потом применим *мемоизацию*

```haskell
fib :: (Int -> Int) -> Int -> Int
fib f 0 = 0
fib f 1 = 1
fib f n = f (n - 1) + f (n - 2)
```

Потом применим *неподвижную точку* (см. слайды про λ-исчисление) и функцию `memoize`:

```haskell
import Data.Function (fix)

fix :: (a -> a) -> a

fibMemo :: Int -> Integer
fibMemo = fix (memoize . fib)
```

Вполне себе асимптотически эффективная функция вычисления чисел Фибоначчи

А если без озарения, то...

Попробуем выжать максимум синтаксическими преобразованиями
Заметим, что вызов функции неявно строит некоторый *констекст*, в котором результат исполнения надо обрабатывать.
Будем передавать контекст явно, называя это *продолжением (continuation)* выполнения функции.

Схема преобразования:

- Пусть изначально есть функция `:: A -> B`
- Мы пишем функцию с типом `:: A -> (B -> r) -> r`, где `r` – новое свежее имя типовой переменной. (`r` – result)
- В тело функции добавляется новый аргумент `k` – продолжение (continuation), с семантикой "то, что надо запустить, когда функция досчитает до ответа".
- Вставляется вызов `k`, там, где исходная функция возвращала результат.

```haskell
fib :: Int -> (Int -> r) -> r
fib 0 k = k 0  -- первые два случая простые
fib 1 k = k 1
fib n k = f (n-1) (\l -> f (n-2) (\r -> k (l+r) )
```

Ну или чуть более читаемо:

```haskell
fib n k =
  f (n-1) $ \l ->
  f (n-2) $ \r ->
  k (l+r)
```

Такой способ программирования называется *continuation passing style*

Ещё примеры

```haskell
mysqrt :: Floating a => a -> a
mysqrt a = sqrt a
print (mysqrt 4) :: IO ()

mysqrt 4 + 2 :: Floating a => a
```

```haskell
mysqrtCPS :: a -> (a -> r) -> r
mysqrtCPS a k = k (sqrt a)
mysqrtCPS 4 print :: IO ()

mysqrtCPS 4 (+ 2) :: Floating a => a
```

```haskell
fac :: Integral a => a -> a

fac 0 = 1
fac n'@(n + 1) =
  n' * fac n
fac 4  :: Integral a => a
```

```haskell
facCPS :: a -> (a -> r) -> r

facCPS 0 k = k 1
facCPS n'@(n + 1) k =
  facCPS n $ \ret -> k (n' * ret)
facCPS 4 id :: Integral a => a
```

Обратите внимание на `id` в случаях, когда с результатом ничего делать не нужно.

Continuation monad

```
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
-- r is the final result type of the whole computation

instance Monad (Cont r) where
  return a       = Cont $ \k -> k a
    -- i.e. return a = \k -> k a
  (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)
    -- i.e. c >>= f = \k -> c (\a -> f a k)
```

С помощью этого можно эмулировать выход из цикла

Continuation monad

```haskell
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }
-- r is the final result type of the whole computation

instance Monad (Cont r) where
  return a       = Cont $ \k -> k a
    -- i.e. return a = \k -> k a
  (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)
    -- i.e. c >>= f = \k -> c (\a -> f a k)
```

С помощью этого можно эмулировать выход из цикла

```haskell
class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a

instance MonadCont (Cont r) where
    callCC f = Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

Длинный пример (1/2)

```
{- We use the continuation monad to perform "escapes" from code blocks.
   This function implements a complicated control structure to process
   numbers:

   Input (n)       Output                       List Shown
   =========       ======                       =========
   0-9             n                            none
   10-199          number of digits in (n/2)    digits of (n/2)
   200-19999       n                            digits of (n/2)
   20000-1999999   (n/2) backwards              none
   >== 2000000      sum of digits of (n/2)       digits of (n/2)
-}
```

Длинный пример (2/2)

```
fun :: Int -> String
fun n = (`runCont` id) $ do
        str <- callCC $ \exit1 -> do                 -- define "exit1"
          when (n < 10) (exit1 (show n))
          let ns = map digitToInt (show (n `div` 2))
          n' <- callCC $ \exit2 -> do                -- define "exit2"
            when ((length ns) < 3) (exit2 (length ns))
            when ((length ns) < 5) (exit2 n)
            when ((length ns) < 7) $
              do let ns' = map intToDigit (reverse ns)
                 exit1 (dropWhile (=='0') ns')      -- escape 2 levels
            return $ sum ns
          return $ "(ns = " ++ (show ns) ++ ") " ++ (show n')
        return $ "Answer: " ++ str
```

Безымянное представление через индексы де Брёйна (de Bruijn)

Идея

- Заводим глобальный контекст Γ, где взаимно однозначно сопоставляем каждому натуральному числу имея переменной.
- Связанные переменные представляем числом $k > 0$. Оно означает, что переменная связывается k-й охватывающей лямбдой.
- Свободная переменная x представляются в виде суммы Γx и глубины её местоположения внутри терма в λ абстракциях.

Пример: $\Gamma = \{b \mapsto 0, a \mapsto 1, z \mapsto 2, y \mapsto 3, x \mapsto 4\}$

- $x(yz) \equiv 4(3\ 2)$
- $(\lambda w \to yw) \equiv (\lambda \to 4\ 0)$
- $(\lambda w \to yx) \equiv (\lambda \to \lambda \to 6)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (1/2)

Пример: $[1 \mapsto s](\lambda \to 2) = [x \mapsto s](\lambda y \to x)$

Когда s проникнет под абстракцию, то надо будет "сдвинуть" некоторые индексы переменных, но не все, например, если $s = 2(\lambda \to 0)$ (т.е. $s = z(\lambda w \to w)$), то надо сдвинуть 2, а не 0.

Определение

Сдвиг терма t на d позиций с отсечкой c (обозначается $\uparrow_c^d (t)$)

- $\uparrow_c^d (k) = \begin{cases} k, & \text{если } k < c \\ k + d, & \text{если } k \geq c \end{cases}$
- $\uparrow_c^d (\lambda \to t_1) = \lambda \to \uparrow_{1+c}^d (t_1)$
- $\uparrow_c^d (t_1 t_2) = \uparrow_c^d (t_1) \uparrow_c^d (t_2)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (2/2)

Определение

Подстановка терма s вместо переменной номер j (обозначается $[j \mapsto s]t$)

- $[j \mapsto s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{в противном случае} \end{cases}$
- $[j \mapsto s](\lambda \to t_1) = (\lambda \to [(j+1) \mapsto \uparrow^1_0 s]t_1)$
- $[j \mapsto s](t_1 t_2) = ([j \mapsto s]t_1 [j \mapsto s]t_2)$

Упражнения на подстановку с индексами де Брёйна

- $[b \mapsto a](b(\lambda x \to \lambda y \to b))$
- $[b \mapsto a(\lambda z \to a)](b(\lambda x \to b))$
- $[b \mapsto a](\lambda b \to b\ a)$
- $[b \mapsto a](\lambda a \to b\ a)$

Индексы де Брёйна и GADT (1/2)

Выражения параметризованы окружением (environment), где хранится информация о введенных переменных, и типом самого выражения

```
data Exp e a where
  Con :: Int                          -> Exp e Int
  Add :: Exp e Int -> Exp e Int       -> Exp e Int
  Var :: Var e a                      -> Exp e a
  Abs :: Typ a -> Exp (e,a) b         -> Exp e (a -> b)
  App :: Exp e (a -> b) -> Exp e a -> Exp e b

data Typ a where
  Int ::                          Typ Int
  Arr :: Typ a -> Typ b -> Typ (a -> b)
```

Типами могут выступать либо `Int`, либо функция из одного типа в другой

Индексы де Брёйна и GADT (2/2)

Окружение – левоориентированно вложенные пары

```
data Env e where
  Emp :: Env ()
  Ext :: Env e -> Typ a -> Env (e,a)
```

Тип переменной в окржунии – это либо вторая компонента, либо что-то вложенное один или более раз

```
data Var e a where
  Zro :: Var (e,a) a
  Suc :: Var e a -> Var (e,b) a
```

Извлекать тип переменной из окружения мы можем, только если типы переменной и окружения согласованы

```
get :: Var e a -> e -> a
get Zro     (_ ,x)     = x
get (Suc n) (xs,_)     = get n xs
```

SKI-комбинаторы – ещё один способ избавиться от имён

SKI-комбинаторы.

Правила редукции:

- $Ix \rightsquigarrow x$
- $Kyx \rightsquigarrow y$
- $Sfgx \rightsquigarrow fx(gx)$

Правила наивного (квадратичного[7]) преобразования:

- $\lambda x \to x \mapsto I$
- $\lambda x \to e \mapsto Ke$, если e – комбинатор или переменная отличная от 1-й
- $\lambda x \to e_1 e_2 \mapsto S(\lambda x \to e_1)(\lambda x \to e_2)$

SKI-комбинаторы можно использовать как представление внутри компилятора

Ссылки I

- Демки на Haskell
  Gitlab repo
- Glamorous Glambda interpreter
  Richard Eisenberg
  github repo
  YouTube Video
- All about monads
  Haskell wiki
- “COMPILING WITH CONTINUATIONS“ book
  Andrew W. Appel
- A Type-Preserving Compiler in Haskell
  Louis-Julien Guillemette & Stefan Monnier
  PDF

Ссылки II

Different folds
Haskell wiki

λ to SKI, Semantically (Declarative Pearl)
Oleg Kiselyov
PDF