

Ещё несколько коротких тем

Косарев Дмитрий а.к.а. Kakadu

матмех СПбГУ

21 ноября 2019 г.

В этих слайдах

1. Безымянные представления

- Индексы де Брёйна
- Индексы де Брёйна и GADT
- SKI

2. Free монада

- Мотивация
- Определение
- Законы Free монад
- Пример. Эмулируем Concurrency
- Про название

Безымянное представление через индексы де Брёйна (de Bruijn)

Идея

- Заводим глобальный контекст Γ , где взаимно однозначно сопоставляем каждому натуральному числу имя 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: $\Gamma = \{b \mapsto 0, a \mapsto 1, z \mapsto 2, y \mapsto 3, x \mapsto 4\}$

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

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

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

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

Определение

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

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

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

Определение

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

- $[j \mapsto s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{в противном случае} \end{cases}$
- $[j \mapsto s](\lambda \rightarrow t_1) = (\lambda \rightarrow [(j+1) \mapsto \uparrow_0^1 s]t_1)$
- $[j \mapsto s](t_1 t_2) = ([j \mapsto s]t_1 [j \mapsto s]t_2)$

Подробнее читать в [3] (Глава 6: Представление термов без использования имён)

Пример

$$(\lambda a \rightarrow \lambda b \rightarrow ab) \ y = (\lambda \rightarrow \lambda \rightarrow 1 \ 0) \ 3$$

$$= [0 \mapsto 3](\lambda \rightarrow 1 \ 0)$$

Свободным переменным нужно увеличить номер

$$= \lambda \rightarrow [1 \mapsto 4](1 \ 0)$$

$$= \lambda \rightarrow ([1 \mapsto 4]1)([1 \mapsto 4]0)$$

$$= \lambda \rightarrow 4 \ 0 =$$

$$\lambda b \rightarrow y \ b =$$

Упражнения на подстановку с индексами де Брёйна

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

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 окружении – 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[4]) преобразования:

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

SKI-комбинаторы можно использовать как представление внутри компилятора

Пример трансляции λ -выражения в SKI

Правила такие:

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

$$A = \lambda x \rightarrow \lambda y \rightarrow yx$$

$$\stackrel{S}{=} \lambda x \rightarrow S(\lambda y \rightarrow y)(\lambda y \rightarrow x)$$

$$\stackrel{I}{=} \lambda x \rightarrow (SI)(\lambda y \rightarrow x)$$

$$\stackrel{K}{=} \lambda x \rightarrow (SI)(Kx)$$

$$\stackrel{S}{=} S(\lambda x \rightarrow SI)(\lambda x \rightarrow Kx)$$

$$\stackrel{S}{=} S(S(\lambda x \rightarrow S)(\lambda x \rightarrow I))(\lambda x \rightarrow Kx)$$

$$= \dots$$

$$\dots \stackrel{K}{=} S(S(KS)(\lambda x \rightarrow I))(\lambda x \rightarrow Kx)$$

$$\stackrel{I}{=} S(S(KS)(KI))(\lambda x \rightarrow Kx)$$

$$\stackrel{S}{=} S(S(KS)(KI))(S(\lambda x \rightarrow K)(\lambda x \rightarrow x))$$

$$\stackrel{K}{=} S(S(KS)(KI))(S(KK)(\lambda x \rightarrow x))$$

$$\stackrel{I}{=} S(S(KS)(KI))(S(KK)I)$$

Очередной мини язык

- `output b` – печатает `"b"`
- `bell` – звенеть как `echo -e "\a"`
- `done` – конец исполнения

Заведем тип для программы, предварительно вот так:

```
data Toy b next = Output b next
                | Bell next
                | Done
```

`Done` всегда последняя и будет означать конец исполнения

Теперь мы умеем конструировать программы

```
data Toy b next = Output b next
                  | Bell next
                  | Done

-- output 'A'
-- done
Output 'A' Done
      :: Toy Char (Toy a next)

-- bell
-- output 'A'
-- done
Bell (Output 'A' Done)
      :: Toy a (Toy Char (Toy b next)))
```

Проблема: для разных программ разный тип.

Применим чит

У нас раньше был комбинатор неподвижной точки *для функций* $y \ f = f \ (y \ f)$, а теперь нам нужно применить комбинатор неподвижной точки *для типов*.

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Теперь у нас программы будут типизироваться всегда одинаково

```
Fix (Output 'A' (Fix Done))  
  :: Fix (Toy Char)
```

```
Fix (Bell (Fix (Output 'A' (Fix Done))))  
  :: Fix (Toy Char)
```

Уже лучше, но есть другая проблема. Программы на мини-языке надо писать *до конца*.

Поведение при окончании программы

Будем в "кидать исключение", когда исполнение должно закончиться

```
data FixE f e = Fix (f (FixE f e)) | Throw e
```

А те, кто исполняют программу, будут исключения "ловить"

```
catch :: (Functor f) => FixE f e1 -> (e1 -> FixE f e2) -> FixE f e2
```

```
catch (Fix x) f = Fix (fmap (flip catch f) x)
```

```
catch (Throw e) f = f e
```

Нам будет нужен функтор.

```
instance Functor (Toy b) where
```

```
    fmap f (Output x next) = Output x (f next)
```

```
    fmap f (Bell      next) = Bell      (f next)
```

```
    fmap f Done          = Done
```

```

data IncompleteException = IncompleteExc

-- output 'A'
-- throw IncompleteExc
subroutine = Fix (Output 'A' (Throw IncompleteExc))
             :: FixE (Toy Char) IncompleteException

-- try { subroutine }
-- catch (IncompleteExc) {
--     bell
--     done
-- }

program :: FixE (Toy Char) e
program = subroutine `catch`
        (\_ -> Fix (Bell (Fix Done)))
        :: FixE (Toy Char) e

```


Проблемка

```
data FixE f e = Fix (f (FixE f e)) | Throw e
```

Мы зарелизили библиотеку, но пользователи используют **Throw**, чтобы передавать нормальные результаты программ, а не моделировать исключительные ситуации. Наверное потому, что мы **навелосипедили** опять!

```
data Free f r = Free (f (Free f r)) | Pure r
```

Ну, вы знаете, что я сейчас скажу

```
data Free f r = Free (f (Free f r)) | Pure r

instance (Functor f) => Monad (Free f) where
    return = Pure
    (Free x) >>= f = Free (fmap (>>= f) x)
    (Pure r) >>= f = f r
```

- `return` \sim `Pure`
- `(>>=)` \sim `catch`

И так как это монада, то у нас будет do-нотация за бесплатно (англ. for free).

Немного сахара для конструирования программ

```
output :: a -> Free (Toy a) ()  
output x = Free (Output x (Pure ()))
```

```
bell :: Free (Toy a) ()  
bell = Free (Bell (Pure ()))
```

```
done :: Free (Toy a) r  
done = Free Done
```

```
liftF :: (Functor f) => f r -> Free f r  
liftF command = Free (fmap Pure command)
```

```
output x = liftF (Output x ())  
bell      = liftF (Bell      ())  
done      = liftF  Done
```

Здесь мы заводим умные кон-
структоры, чтобы писать меньше
boilerplate кода

Всё просто работает

Раньше монады использовались только чтобы имитировать эффекты, а теперь – чтобы *конструировать* данные.

```
subroutine :: Free (Toy Char) ()
```

```
subroutine = output 'A'
```

```
program :: Free (Toy Char) r
```

```
program = do
```

```
    subroutine
```

```
    bell
```

```
    done
```

Покажем, что это действительно данные, написав интерпретатор, который распечатывает

```
showProgram :: (Show a, Show r) => Free (Toy a) r -> String
```

```
showProgram (Free (Bell x)) = "bell\n" ++ showProgram x
showProgram (Free Done)     = "done\n"
showProgram (Pure r)        = "return " ++ show r ++ "\n"
showProgram (Free (Output a x)) =
    "output " ++ show a ++ "\n" ++ showProgram x
```

Распечатаем программу:

```
>>> putStr (showProgram program)
output 'A'
bell
done
```

А что там с законами монад? (1/3)

- Левый нейтральный: $\text{return } a \gg= f \equiv f \ a$
- Правый нейтральный: $m \gg= \text{return} \equiv m$
- Ассоциативность (почти): $(m \gg= f) \gg= g \equiv m \gg= (\backslash x \rightarrow f \ x \gg= g)$

```
pretty :: (Show a, Show r) => Free (Toy a) r -> IO ()
pretty = putStr . showProgram
```

```
✓ >>> pretty (output 'A')
    output 'A'
    return ()
```

А что там с законами? (2/3)

? >>> pretty (return 'A' >>= output)

А что там с законами? (2/3)

? >>> pretty (return 'A' >>= output)

output 'A'

return ()

? >>> pretty (output 'A' >>= return)

А что там с законами? (2/3)

? >>> pretty (return 'A' >>= output)

output 'A'

return ()

? >>> pretty (output 'A' >>= return)

output 'A'

return ()

А что там с законами? (3/3)

? >>> pretty ((output 'A' >> done) >> output 'C')

А что там с законами? (3/3)

? >>> pretty ((output 'A' >> done) >> output 'C')

output 'A'

return ()

? >>> pretty (output 'A' >> (done >> output 'C'))

А что там с законами? (3/3)

? >>> pretty ((output 'A' >> done) >> output 'C')
output 'A'
return ()

? >>> pretty (output 'A' >> (done >> output 'C'))
output 'A'
return ()

Можно и нормальный интерпретатор

Будем использовать функцию `ringbell` из сторонней библиотеки, которая на самом деле будет "звенеть".

```
ringBell :: IO ()
```

```
interpret :: (Show b) => Free (Toy b) r -> IO ()
```

```
interpret (Free (Output b x)) = print b >> interpret x
```

```
interpret (Free (Bell      x)) = ringBell >> interpret x
```

```
interpret (Free Done        ) = return ()
```

```
interpret (Pure r) =
```

```
  throwIO (userError "Unexpected termination")
```

Для Free монады всё равно как вы её используете.

Если хотим concurrency для монады IO, то можно вызывать `forkIO` из модуля `Control.Concurrent`.

А если хотим это сделать для других монад: `State` или `Cont`?

Первое желание: список монадических "действий".

```
type Thread m = [m ()]
```

Но

- Тут нет информации о порядке вызовов
- И непонятно, где тут результат работы.

Пример про кооперативную многозадачность

Один конструктор, чтобы интерпретатор знал что за чем делать. Второй для результата

```
data Thread m r = Atomic (m (Thread m r)) | Return r
```

Вычисление одного шага будет выглядеть так:

```
atomic :: (Monad m) => m a -> Thread m a
```

```
atomic m = Atomic (liftM Return m)
```

где стандартная функция `liftM` работает как `fmap`

```
liftM :: Monad m => (a1 -> r) -> m a1 -> m r
```

Очевидно, что этот тип отличается от `Free` только переименовыванием конструкторов:

`Thread` – это `Free`, а `atomic` – `liftF`.

Сконструируем пару кооперативных задач...

```
thread1 :: Thread IO ()
thread1 = do
    atomic (print 1)
    atomic (print 2)

thread2 :: Thread IO ()
thread2 = do
    str <- atomic getLine
    atomic (putStrLn str)
```

```
interleave :: (Monad m) =>
    Thread m r -> Thread m r -> Thread m r

interleave (Atomic m1) (Atomic m2) = do
    next1 <- atomic m1
    next2 <- atomic m2
    interleave next1 next2

interleave t1 (Return _) = t1
interleave (Return _) t2 = t2
```


.... и запустим их

```
runThread :: (Monad m) => Thread m r -> m r
runThread (Atomic m) = m >>= runThread
runThread (Return r) = return r
```

```
>>> runThread (interleave thread1 thread2)
```

```
1
```

```
[[Input: 'Hello, world!']]
```

```
2
```

```
Hello, world!
```

Здесь в квадратных скобках – то, что вводилось человеком с клавиатуры.

Тип `Thread` сильно напоминает список

А пример с потоками говорит, что `Free` жутко напоминает `List`.

```
data Free f r = Free (f (Free f r)) | Pure r
data List a   = Cons a (List a )   | Nil
```

В некотором смысле, `Free` – это список (`List`) функторов.

Определение (Интуиция за **Free** монадой)

Free f a – это тип деревьев с формой f и листьями типа a . Операция объединения соединяет деревья не проводя никаких вычислений

Пример: **Free Unit** изоморфно монаде **Maybe**

```
data Unit a = MkUnit
```

Если мы подставим определение **Unit** во **Free** вместо f , то мы получим два случая:
Free Unit соответствует **Nothing** (так как у деревьев с корнем вида **Free** нет листьев), а
Pure a – **Just** a

Попробуем выразить список через Free

```
data Free f r = Free (f (Free f r)) | Pure
```

```
type List' a = Free ((,) a) ()
```

Проверим изоморфность:

```
List' a = Free ((,) a) ()  
        = Free (a, List' a) | Pure ()  
        ~ Free a (List' a) | Pure ()  
        ~ Cons a (List' a) | Nil
```

Упражнение

У нас был instance монады для Free. Следовательно, у нас есть инстанс для `type List' a = Free ((,) a) ()`





И ещё мы знаем, что `[]` это тоже монада.

Упражнение

? Одинаковые ли ведут себя инстансы для двух типов:

- `type List' a = Free ((,) a) ()` и
- `[]`

т.е. одинаковые ли это монады? Если да, то почему. Если нет, то предъявите программы, где они ведут себя по-разному.

-  Демки на Haskell
[Gitlab repo](#)
-  Glamorous Glambda interpreter
Richard Eisenberg
[github repo](#)
[YouTube Video](#)
-  Бенджамин Пирс
Типы в языках программирования. (Первый том)
-  λ to SKI, Semantically (Declarative Pearl)
Oleg Kiselyov
[PDF](#)



Gabriel Gonzalez

Why free monads matter

[ссылка](#)



Janis Voigtländer

Asymptotic Improvement of Computations over Free Monads

[paper](#), [presentation](#)