# CS314. Функциональное программирование

## Лекции 16–17. Семейства типов

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

24 ноября 2017 г.

# Содержание

# Содержание

# Проблема: смешивание числовых типов в вычислении

```
ghci> 5 + 7
12
ghci> 5 + 7.0
12.0
ghci> (5 :: Int) + 7.0
<interactive>:4:14:
 No instance for (Fractional Int) arising from the literal ‘7.0’
 In the second argument of ‘(+)’, namely ‘7.0’
 In the expression: (5 :: Int) + 7.0
 In an equation for ‘it’: it = (5 :: Int) + 7.0

ghci> (5 :: Int) + (7.0 :: Double)
<interactive>:6:15:
 Couldn't match expected type ‘Int’ with actual type ‘Double’
 In the second argument of ‘(+)’, namely ‘(7.0 :: Double)’
 In the expression: (5 :: Int) + (7.0 :: Double)
 In an equation for ‘it’: it = (5 :: Int) + (7.0 :: Double)
```

# Способы решения

## Явное приведение типа

```
ghci> fromIntegral (5 :: Int) + (7.0 :: Double)
12.0
```

```
fromIntegral :: (Integral a, Num b) => a -> b
```

## Важные соображения

- Хорошая система типов не должна отвергать «хорошие» программы!
- Всем ясно, что Int + Double должно быть Double!

# Обобщение класса типов Num

```
{-# LANGUAGE MultiParamTypeClasses #-}

class GNum a b where
  plus :: a -> b -> ???

instance GNum Int Int where
  plus a b = a + b

instance GNum Int Double where
  plus a b = fromIntegral a + b
```

```
ghci> plus (5 :: Int) (7.0 :: Double)
12.0
```

- Но что делать с ???
- По идее тип нужно «вычислять».

# Семейства типов

```haskell
{-# LANGUAGE MultiParamTypeClasses, TypeFamilies #-}

class GNum a b where
  type SumTy a b :: *
  plus :: a -> b -> SumTy a b

instance GNum Int Int where
  type SumTy Int Int = Int
  plus a b = a + b

instance GNum Int Double where
  type SumTy Int Double = Double
  plus a b = fromIntegral a + b
```

- `SumTy :: * -> * -> *` — функция на уровне типов (type-level function, семейство типов, индексированное семейство типов)

# Содержание

# Виды и способы объявления семейств типов

## Виды семейств типов

- Открытые семейства синонимов типов (open type synonym families)
- Замкнутые семейства синонимов типов (closed type synonym families)
- Семейства типов данных (data families)

## Способы объявления семейств типов

- На верхнем уровне
- Внутри классов типов (ассоциированные типы) — кроме замкнутых семейств синонимов типов

```haskell
{-# LANGUAGE MultiParamTypeClasses, TypeFamilies #-}

class GNum a b where
  type SumTy a b :: *
  plus :: a -> b -> SumTy a b

instance GNum Int Int where
  type SumTy Int Int = Int
  plus a b = a + b

instance GNum Int Double where
  type SumTy Int Double = Double
  plus a b = fromIntegral a + b
```

- SumTy — ассоциированный тип (открытое семейство синонимов, объявленное внутри класса типов)

# Открытые и замкнутые семейства синонимов

## Открытое семейство

```
type family Elem c :: *
type instance Elem [e] = e
type instance Elem (Set a) = a
type instance Elem (Seq a) = a
```

- Экземпляры можно дописывать в любом месте

## Замкнутое семейство

```
type family F t where
  F Int = Bool
  F Bool = Int
  F Char = String
```

- Можно объявлять только на верхнем уровне
- Расширить замкнутое семейство в другом месте нельзя

# Использование замкнутых семейств типов

```
type family F a where
  F Int = Bool
  F Bool = Int
  F Char = String
```

**Как определить функцию convert?**

```
convert :: a -> F a
```

```
ghci> convert 'x'
"x"
ghci> convert (42 :: Int)
True
ghci> convert False
0
```

- Проблема: сопоставление с образцом не позволяет различать типы

# Использование замкнутых семейств типов

- Класс типов необходим всё равно:

```
class Convertible a where
  convert :: a -> F a

instance Convertible Int where
  convert 0 = False
  convert n = True

instance Convertible Bool where
  convert True = 1
  convert False = 0

instance Convertible Char where
  convert c = [c]
```

# Семейства типов данных (на верхнем уровне)

```
data family T a :: *
data instance T Int  = T1 Int | T2 Bool
data instance T Char = TC Bool
```

- T1, T2, TC — объявляемые конструкторы данных
- Семейства типов данных открыты (в любой момент и в любом другом модуле можно дописать значение для нового типа)
- Семейства типов данных инъективны (конструкторы данных в результатах обязательно различны)

## Можно ли написать такую функцию?

```
data family T a
data instance T Int  = A
data instance T Char = B

foo :: T a -> Int
foo A = ...
foo B = ...
```

- Нет, этот код приведёт к ошибке типизации.
- Для объявления функций нужен класс типов и его экземпляры:

```
class Foo a where
  foo :: T a -> Int
instance Foo Int where
  foo A = ...
instance Foo Char where
  foo B = ...
```

# Ассоциированные типы данных

```haskell
class Foo a where
  data family T a
  foo :: T a -> Int

instance Foo Int where
  data T Int = A
  foo A = ...

instance Foo Char where
  data T Char = B
  foo B = ...
```

# Содержание

# Содержание

# Изменяемые хранилища на семействах типов

```
class Store m where
  type Ref m :: * -> *
  new :: a -> m (Ref m a)
  get :: (Ref m a) -> m a
  put :: (Ref m a) -> a -> m ()
```

- Вид ссылки на изменяемую переменную `Ref m a` зависит от используемой монады и типа элемента

```
instance Store IO where
  type Ref IO = IORef
  new = newIORef
  get = readIORef
  put = writeIORef
```

```
instance Store (ST s) where
  type Ref (ST s) = STRef s
  new = newSTRef
  get = readSTRef
  put = writeSTRef
```

# Содержание

# Что такое граф?

```
class Graph g where
  type Vertex g :: *
  data Edge g :: *
  src, tgt :: Edge g -> Vertex g
  outEdges :: g -> Vertex g -> [Edge g]
  -- ...другие методы...
```

# Различные представления

## Список дуг

```
newtype G1 = G1 [Edge G1]

instance Graph G1 where
  type Vertex G1 = Int
  data Edge   G1 = MkEdge1 (Vertex G1) (Vertex G1)
  -- ...определения методов...
```

## Списки смежных вершин

```
newtype G2 = G2 (Map (Vertex G2) [Vertex G2])

instance Graph G2 where
  type Vertex G2 = String
  data Edge G2 = MkEdge2 Int (Vertex G2) (Vertex G2)
  -- ...определения методов...
```

# Использование гарантированной инъективности Edge

- Каков тип функции?

```
neighbours g v = map tgt (outEdges g v)
```

- Компилятор видит, что мы достаём дуги из графа g, а потом извлекаем из них смежные с v вершины — функции outEdges и tgt связаны через дуги:

```
outEdges :: g -> Vertex g -> [Edge g]
tgt :: Edge g -> Vertex g
```

- Проблема в том, что outEdges и tgt — обобщённые функции, они могут работать с одинаковыми дугами даже из разных графов!
- Инъективность гарантирует, что если дуги совпадают, то совпадают и графы, поэтому проблем не будет:

```
neighbours :: Graph g => g -> Vertex g -> [Vertex g]
```

# Содержание

# Как реализовать ассоциативный контейнер (Map)?

- Ассоциативный контейнер = отображение из ключей в значения
- Эффективность реализации зависит от типа ключа:
  - Bool — просто ленивая пара
  - Int — сбалансированное дерево поиска
  - Пара (a, b) — отображение по a на отображения по b
  - Список [a] — рекурсивная последовательность отображений
- Почему бы не использовать семейство типов данных, индексированное типом ключа?

# Ключ и семейство зависящих от него представлений ассоциативного контейнера

```
class Key k where
  data Map k :: * -> *
  empty :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ... insert, union и т.д. ...
```

- Первая звёздочка в *->* соответствует типу значения, а вторая — типу получающегося в результате контейнера

# Различные реализации

## Ключ как Bool

```
instance Key Bool where
  data Map Bool elt = MB (Maybe elt) (Maybe elt)
  empty = MB Nothing Nothing
  lookup False (MB mf _) = mf
  lookup True (MB _ mt) = mt
  ...
```

## Ключ как пара

```
instance (Key a, Key b) => Key (a,b) where
  data Map (a,b) elt = MP (Map a (Map b elt))
  empty = MP empty
  lookup (a,b) (MP m) = case lookup a m of
                          Nothing -> Nothing
                          Just m' -> lookup b m'
  ...
```

# Различные реализации (2)

## Ключ как список

```
instance (Key a) => Key [a] where
  data Map [a] v = ML (Maybe v) (Map (a,[a]) v)
  empty = ML Nothing empty
  lookup [] (ML m0 _) = m0
  lookup (h:t) (ML _ m1) = lookup (h,t) m1
  ...
```

## Экземпляр для конкретного типа

```
instance Key Int where
  data Map Int elt = IM (Data.IntMap.Map elt)
  empty = IM Data.IntMap.empty
  lookup k (IM m) = Data.IntMap.lookup m k
  ...
```

# Пример использования различных реализаций

## Экземляр для Bool

```
ghci> let mb = insert True 1 empty
ghci> lookup True mb
Just 1
ghci> lookup False mb
Nothing
```

## Экземпляр для Int

```
ghci> let mi = insert (5 :: Int) "hello" empty
ghci> lookup 5 mi
Just "hello"
ghci> lookup 10 mi
Nothing
```

# Содержание

# Проблема

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

- Функция `fib` вычисляет одни и те же значения много раз.
- Мемоизацией функции называется сохранение ранее вычисленных значений в специальной таблице, индексированной аргументом функции.
- Благодаря ленивости вычисляться будут только реально запрошенные значения.

# Общий подход к мемоизации функций

## Класс Memo

```
class Memo a where
  data Table a :: * -> *
  toTable :: (a -> w) -> Table a w
  fromTable :: Table a w -> (a -> w)
```

## Пример использования

```
fib :: Integer -> Integer
fib = fromTable (toTable fib')
  where
    fib' 0 = 1
    fib' 1 = 1
    fib' n = fib (n-1) + fib (n-2)
```

# Реализация Memo для простых типов

## Memo для Bool

```
instance Memo Bool where
  data Table Bool w = TBool w w
  toTable f = TBool (f True) (f False)
  fromTable (TBool x y) b = if b then x else y
```

## Memo для ()

```
instance Memo () where
  newtype Table () w = TUnit w
  toTable f = TUnit (f ())
  fromTable (TUnit x) () = x
```

# Реализация Memo для составных типов

## Memo для пары

```
instance (Memo a, Memo b) => Memo (a,b) where
  newtype Table (a,b) w =
                  TProduct (Table a (Table b w))
  toTable f = TProduct (toTable (\x -> toTable
                                 (\y -> f (x,y))))
  fromTable (TProduct t) (x,y) =
                        fromTable (fromTable t x) y
```

## Memo для списка

```
instance (Memo a) => Memo [a] where
  data Table [a] w = TList w (Table a (Table [a] w))
  toTable f = TList (f [])
              (toTable $ \x -> toTable $
                                \xs -> f (x:xs))
  fromTable (TList t _) []     = t
  fromTable (TList _ t) (x:xs) =
              fromTable (fromTable t x) xs
```

# Memo для Integer

```
instance Memo Int where
  data Table Int w = TInt (Table [Bool] w)
  toTable f = TInt $ toTable $ \bs ->
                                 f $ bitsToInt bs
  fromTable (TInt t) n = fromTable t (intToBits n)
```

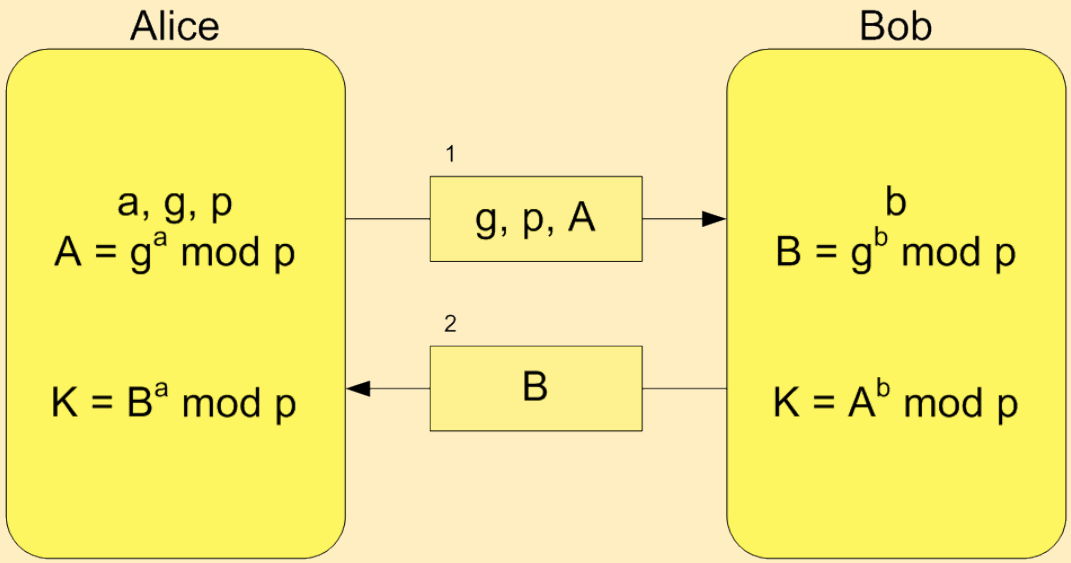
# Содержание

# Сеансы и протоколы

## Протокол суммирования

- Клиент последовательно посылает серверу два числа.
- Сервер получает числа и отправляет обратно их сумму.
- Клиент получает и печатает сумму.

## Протокол Диффи-Хеллмана




$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Сеанс — это взаимодействие двух сторон, при котором получаемые и отправляемые данные должны строго друг другу соответствовать.

# Реализация протокола суммирования

## Базовые типы

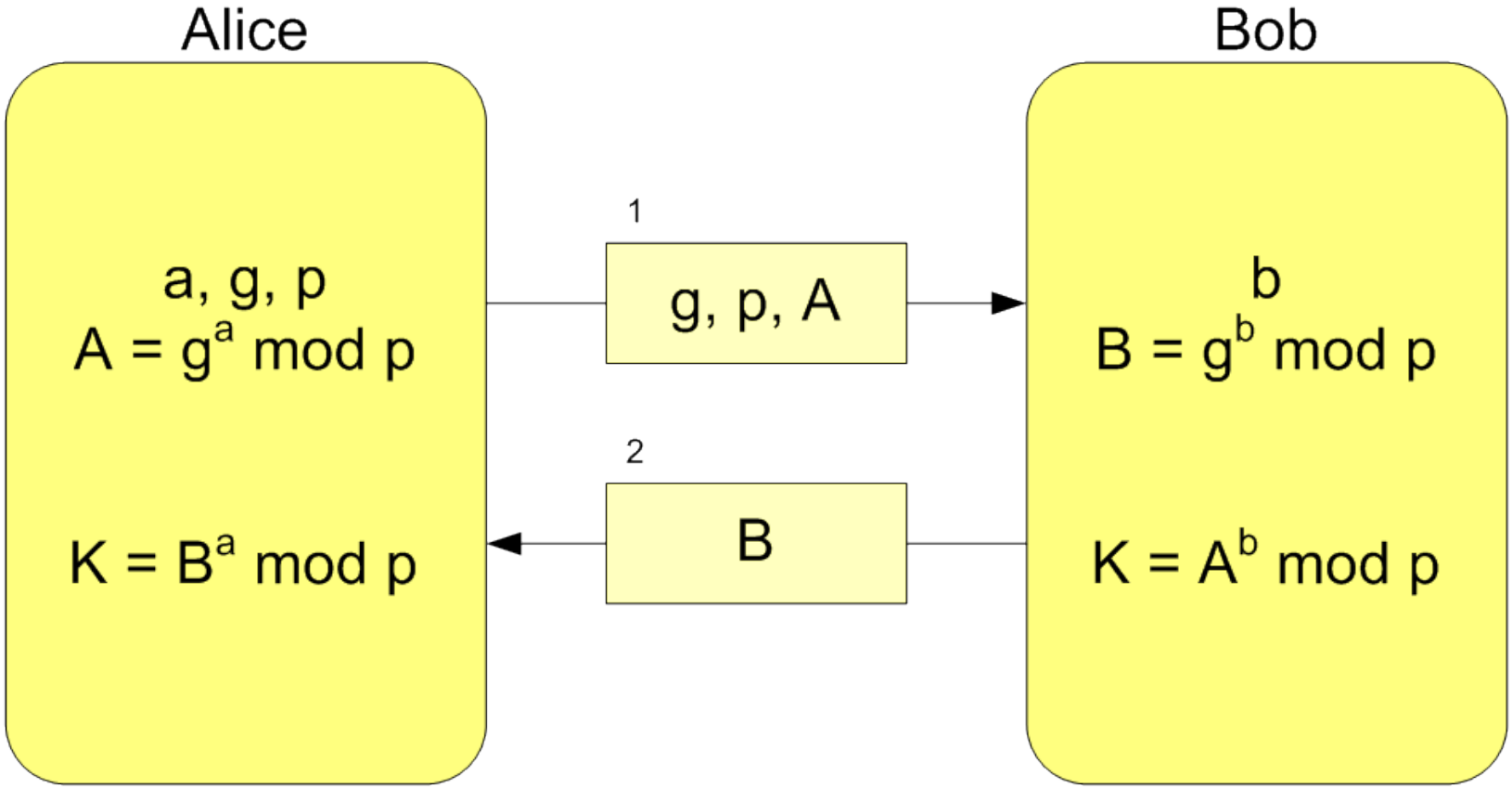
```
data Stop = Done
newtype In  a b = In (a -> IO b)
data    Out a b = Out a    (IO b)
```

## Сервер и клиент протокола суммирования

```
add_server :: In Int (In Int (Out Int Stop))
add_server = In $ \x -> return $ In $ \y -> do
                 putStrLn "Thinking"
                 return $ Out (x + y) (return Done)

add_client :: Int -> Int -> Out Int
                              (Out Int (In Int Stop))
add_client n m = Out n $ return $ Out m $ do
          putStrLn "Waiting"
          return $ In $ \z -> print z >> return Done
```

# Реализация протокола Диффи-Хеллмана




$$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$$

# Реализация протокола Диффи-Хеллмана

## Сервер и клиент протокола Диффи-Хеллмана

```
alice :: Int -> (Int, Int) ->
         Out (Int, Int, Int) (In Int Stop)
alice sk (g, p) = Out (g, p, mod (g^sk) p) $
  return $ In $ \b -> do
    putStrLn $ "Alice's shared key: "
                    ++ show (mod (b^sk) p)
    return Done

bob :: Int -> In (Int, Int, Int) (Out Int Stop)
bob sk = In $ \(g, p, a) -> do
   putStrLn $ "Bob's shared key: "
                        ++ show (mod (a^sk) p)
   return $ Out (mod (g^sk) p) (return Done)
```

# Сеансы

```haskell
class Session a where
  type Dual a
  run :: a -> Dual a -> IO ()
```

```haskell
instance (Session b) => Session (In a b) where
  type Dual (In a b) = Out a (Dual b)
  run (In f) (Out a d) =
              f a >>= \b -> d >>= \c -> run b c
instance (Session b) => Session (Out a b) where
  type Dual (Out a b) = In a (Dual b)
  run (Out a d) (In f) =
              f a >>= \b -> d >>= \c -> run c b
instance Session Stop where
  type Dual Stop = Stop
  run Done Done = return ()
```

# Запуск сеанса: протокол суммирования

```
test1 = run add_server (add_client 3 4)
test2 = run (add_client 3 4) add_server
```

```
ghci> test1
Thinking
Waiting
7
ghci> test2
Thinking
Waiting
7
```

- Проверка типа функции `run` заключается в проверке совместимости двух процессов — клиента и сервера.

# Запуск сеанса: протокол Диффи-Хеллмана

```
test_dh = run (alice 6 (5, 23)) (bob 15)
```

```
ghci> test_dh
Bob's shared key: 2
Alice's shared key: 2
```

# Содержание

# Функция sprintf (C)

```
int sprintf ( char* str, const char* format, ... );
```

```c
#include <stdio.h>

int main ()
{
  char buffer [50];
  int n, a=5, b=3;
  n = sprintf(buffer, "%d plus %d is %d",
                      a, b, a+b);
  printf("[%s] is a string %d chars long\n",
                      buffer, n);
  return 0;
}
```

- Такая функция в Haskell невозможна!

# Но кое-что всё-таки возможно!

```
sprintf :: F f -> SPrintF f
```

```
-- "Day"
sprintf (lit "Day") :: String
-- "Day %n"
sprintf (lit "Day " <> int) :: Int -> String
-- "Day %n Month %s"
sprintf (lit "Day " <> int
                    <> lit "Month "
                    <> string)
                :: Int -> String -> String
```

- Маленький DSL для описания форматов
- Зависимость типа sprintf от типа “форматной строки”

## Типы для форматных строк (метки форматов)

```
data L
data V val
data C a b
```

## DSL для описания формата (GADT)

```
data F f where
  Lit  :: String -> F L
  Val  :: Show a => F (V a)
  Comp :: F f1 -> F f2 -> F (C f1 f2)
```

```
lit :: String -> F L
lit str = Lit str

int :: F (V Int)
int = Val
```

```
string :: F (V String)
string = Val

infixl 5 <>
(<>) = Comp
```

# Примеры типов “форматных строк”

```
ghci> :t lit "Day "
lit "Day " :: F L

ghci> :t lit "Day " <> int
lit "Day " <> int :: F (C L (V Int))

ghci> :t lit "Day " <> int <> lit "Month " <> string
lit "Day " <> int <> lit "Month " <> string
  :: F (C (C (C L (V Int)) L) (V String))

ghci> :t int <> int <> int
int <> int <> int
          :: F (C (C (V Int) (V Int)) (V Int))
```

# Вычисление типа sprintf

```
type family TPrinter f x

type instance TPrinter L          x = x
type instance TPrinter (V val)    x = val -> x
type instance TPrinter (C f1 f2) x =
                    TPrinter f1 (TPrinter f2 x)
type SPrintF f = TPrinter f String
```

```
lit "Day " <> int :: F (C L (V Int))
```

```
SPrintF (C L (V Int))
    === TPrinter (C L (V Int)) String
    === TPrinter L (TPrinter (V Int) String)
    === TPrinter (V Int) String
    === Int -> String
```

# Реализация sprintf

```
printer :: F f -> (String -> a) -> TPrinter f a
printer (Lit str)    k = k str
printer Val          k = \x -> k (show x)
printer (Comp f1 f2) k =
         printer f1 (\s1 ->
                printer f2 (\s2 -> k (s1++s2)))
```

- Continuation-passing style (CPS)

```
sprintf :: F f -> SPrintF f String
sprintf p = printer p id
```

# Пример использования

```
ghci> :t sprintf (lit "Day " <> int)
sprintf (lit "Day " <> int) :: Int -> String
ghci> sprintf (lit "Day " <> int) 28
"Day 28"
```

# Вычисление результата

```
printer :: F f -> (String -> a) -> TPrinter f a
printer (Lit str)    k = k str
printer Val          k = \x -> k (show x)
printer (Comp f1 f2) k = printer f1 (\s1 ->
                           printer f2 (\s2 -> k (s1++s2)))
sprintf :: F f -> SPrintF f String
sprintf p = printer p id
```

```
sprintf (lit "Day " <> int) 28
  === printer (Comp (Lit "Day ") Val) id 28
  === printer (Lit "Day ") (\s1 -> printer Val (\s2 ->
                                    id (s1 ++ s2))) 28
  === (\s1 -> printer Val (\s2 -> id (s1 ++ s2))) "Day " 28
  === printer Val (\s2 -> id ("Day " ++ s2)) 28
  === (\x -> (\s2 -> id ("Day " ++ s2)) (show x)) 28
  === (\s2 -> id ("Day " ++ s2)) (show 28)
  === id ("Day " ++ (show 28)) === "Day " ++ (show 28)
  === "Day " ++ "28" === "Day 28"
```

# Содержание

# Проблема выравнивания указателей

- Доступ к памяти обычно выполняется не по произвольным октетам (байтам), а по выровненным (aligned) адресам, то есть делящимся на 2, 4,...
- Адрес — это просто целое число, но нельзя допускать произвольную совместную работу с указателями, выровненными по-разному.
- Для различения таких целочисленных адресов можно использовать фантомные типы.

# Натуральные числа на уровне типов

## Представление $\mathbb{N}$ на типах

```
data Zero
data Succ n

type One   = Succ Zero
type Two   = Succ One
type Three = Succ Two
type Four  = Succ Three
type Five  = Succ Four
type Six   = Succ Five
type Seven = Succ Six
type Eight = Succ Seven
type Nine  = Succ Eight
type Ten   = Succ Nine
```

## Класс Nat (переход к значениям)

```
class Nat n where
  toInt :: n -> Int

instance Nat Zero where
  toInt _ = 0

instance (Nat n) => Nat (Succ n) where
  toInt _ = 1 + toInt (undefined :: n)
```

- Значения этих типов нас не интересуют, эти типы фантомные.

# Указатели и смещения

```
newtype Pointer n = MkPointer Int
newtype Offset  n = MkOffset  Int
```

- n — фантомный тип (не используется в конструкторах)
- newtype — нет накладных расходов во время выполнения

## Создание указателя

```
pointer :: forall n. (Nat n) => Int -> Pointer n
pointer i
  | mod i (toInt (undefined :: n)) == 0 = MkPointer i
  | otherwise = error "wrong alignment"
```

## Вычисление корректного смещения

```
offset :: forall n. Nat n => Int -> Offset n
offset i = MkOffset (i * toInt (undefined :: n))
```

`Pointer Six + Offset Four`?

# Вычисление наибольшего общего делителя

```
type family GCD d m n          --- gcd (d+m, d+n)
```

```
type instance GCD d Zero Zero = d
type instance GCD d (Succ m) (Succ n) = GCD (Succ d) m n
type instance GCD Zero (Succ m) Zero = Succ m
type instance GCD (Succ d) (Succ m) Zero =
                                    GCD (Succ Zero) d m
type instance GCD Zero Zero (Succ n) = Succ n
type instance GCD (Succ d) Zero (Succ n) =
                                    GCD (Succ Zero) d n
```

```
GCD Zero Six Four
  === GCD One Five Three === GCD Two Four Two
  === GCD Three Three One === GCD Four Two Zero
  === GCD One Three One === GCD Two Two Zero
  === GCD One One One === GCD Two Zero Zero === Two
```

# Сложение указателя и смещения

```
add :: Pointer m -> Offset n -> Pointer (GCD Zero m n)
add (MkPointer x) (MkOffset y) = MkPointer (x + y)
```

- Пользователю этой функции гарантируется, что указатель окажется правильно выровненным.

```
p :: Pointer Six
p = pointer 18

o :: Offset Four
o = offset 2
```

```
ghci> :t add p o
add p o :: Pointer (Succ (Succ Zero))
ghci> add p o
MkPointer 26
```

# Сюрприз: сложение на типах

```
type family Plus m n
type instance Plus Zero n = n
type instance Plus (Succ m) n = Succ (Plus m n)
```

```
ghci> :t (undefined :: Plus Two Three)
(undefined :: Plus Two Three)
                    :: Succ (Succ (Succ Two))
```

# Библиография


1. GHC User's Guide
   https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/
2. Oleg Kiselyov, Simon Peyton Jones, Chung-chieh Shan. Fun with type functions
   http://research.microsoft.com/en-us/um/people/simonpj/papers/assoc-types/fun-with-type-funs/typefun.pdf