Зависимые типы в GHC 8

Максим Талдыкин

max@formalmethods.ru

Что такое?

```
Just 1 : Maybe Int : * : □
Maybe : * -> * : □
Vec Int Z : * : □
Vec : * -> Nat -> * : □
```

```
map : forall a b. (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map f [] = []
map :: forall a b. (a -> b) -> [a] -> [b]
map =
  \ (@a)
    (@b)
    (f :: a -> b)
    (ds :: [a]) ->
    case ds of
      [] -> GHC.Types.[] @b
      : x xs ->
        GHC.Types.: @b (f x) (map @a @b f xs)
```

```
forall
  (a :: *)
  (b :: *)
  (f :: a -> b)
  (xs :: [a])
  . [b]
```

```
replicate :: pi (n : Nat) -> a -> Vec a n
replicate Z     _ = Nil
replicate (S m) x = x :> replicate m a
```

	vis	**dep**	**rel**
forall (a : *). a	-	+	-
a -> a	+	-	+
Monad m => m a	-	-	+
pi (a : Bool). f a	-	+	+
pi (a : Bool) -> f a	+	+	+

* : *

pi

Зачем?

zipWithN, mapT

tail []

```haskell
strncpy
  :: forall (p q :: Nat)
  => (LE n p ~ True, LE n q ~ True)
  .  pi (n :: Nat)
  -> Ptr Char p
  -> Ptr Char q
  -> IO ()
```

- Π-Ware: Hardware Description
    - Swierstra et al., 2015
- Correct-by-Construction Concurrency
    - Brady & Hammond, 2009
- Security-Typed Programming
    - Morgenstern & Licata, 2010

Type Families

```haskell
data Nat = Z | S Nat

type family
    (m :: Nat) :+ (n :: Nat) :: Nat
    where
      Z     :+ n = n
      (S k) :+ n = S (k :+ n)

-- Vec a m -> Vec a n -> Vec a (m :+ n)
```

GADTs

Generalized Algebraic Data Types

```
data Vec :: Nat -> * -> *
  Nil  :: Vec Z a
  (:>) :: a -> Vec n a -> Vec (S n) a
```

```
data Vec (n :: Nat) (a :: *) :: * where
  Nil  :: n ~ Z => Nil n a
  (:>) :: forall (m :: Nat)
       .  n ~ S m
       => a -> Vec m a -> Vec n a
```

```haskell
tail :: Vec (S k) a -> Vec k a
tail (_ :> xs) = xs
  -- (m :: Nat) (S k ~ S m)
```

```
type family
  (v :: Vec a m) :++ (w :: Vec a n) :: Vec a k
  where
    Nil       :++ w = w
    (x :> xs) :++ w = x :> (xs :++ w)
```

```
type family
  (v :: Vec a m) :++ (w :: Vec a n) :: Vec a k
  where
    Nil       :++ w = w
    (x :> xs) :++ w = x :> (xs :++ w)
```

```
‘Vec’ of kind ‘* -> Nat -> *’ is not promotable
In the kind ‘Vec a m’
```

```
Vec :: □ -> □ -> □
```

```
replicate :: pi (n :: Nat) -> a -> Vec a n
replicate Z a = Nil
replicate (S m) a = a :> replicate m a
```

```haskell
data Nat = Z | S Nat

data Nat's :: Nat -> * where
  Z's :: Nat's Z
  S's :: Nat's n -> Nat's (S n)

--  S's Z's :: Nat's (S Z)
```

```
replicate :: Nat's n -> a -> Vec a n
replicate Z's a = Nil
replicate (S's m) a = a :> replicate m a
```

Они уже здесь!

Singleton types here

Singleton types there

Singleton types everywhere

Monnier & Haguenauer, 2009

Hasochism

Lindley & mcBride, 2013

```
data Fin :: Nat -> * where
  FZ :: Fin (S n)
  FS :: Fin n -> Fin (S n)
```

```
lookup :: pi (f :: Fin n) -> Vec a n -> a
lookup FZ (x :> _) = x
lookup (FS i) (_ :> xs) = lookup i xs
```

```
data Fin's (n :: Nat) (f :: Fin n) :: * where
  FZ's :: Fin's (S m) FZ
  FS's :: Fin's m g -> Fin's (S m) (FS g)

-- FS's FZ's :: Fin's 4 (FS FZ :: Fin 4)
```

```
data Fin's (n :: Nat) (f :: Fin n) :: * where
  FZ's :: Fin's (S m) FZ
  FS's :: Fin's m g -> Fin's (S m) (FS g)
```

```
Kind variable also used as type variable: ‘n’
In the data type declaration for ‘Fin's’
```

```haskell
get "/Contract/:id" $ do
  intParam "id" >>= queryDb "Contract" >>= json

type API
  = "Contract"
    :> Capture "id" Int
    :> Get '[JSON] Contract
```

```haskell
type Api
  = "Contract"
    :> Capture "id" Int
    :> RoleFilter "Contract" "owner" '[43, 265]
    :> Get '[JSON] (Obj Contract)
```

```
data User (fieldName :: Symbol) where
  Id :: Int -> User "id"
  Name :: Text -> User "name"
  Roles :: [Vector (Ref Role)] -> User "roles"

data Contract (fieldName :: Symbol) where
  Id :: Int -> Contract "id"
  Owner :: Ref Role -> Contract "owner"

type family TableName (m :: Symbol -> *) :: Symbol
type instance TableName User = "User"
type instance TableName Contract = "Contract"
type Api
  = "Contract"
    :> Capture "id" Int
    :> RoleFilter "Contract" "owner" '[43, 265]
    :> Get '[JSON] (Obj Contract)

type Api
  = MkFilter Contract
    (RoleFilter C.Owner '[Role1, Role2])
```

```haskell
type family MkFilter
  (f :: Symbol)
  (m :: Symbol -> *)
  (filter :: *)
  where
    MkFilter f m (RoleFilter (own :: m g) rs)
      = TableName m
      :> Capture
        (FieldName (TableId m))
        (FieldType (TableId m))
      :> RoleFilter' (TableName m) g rs
      :> Get '[JSON] (Obj m)
```

Termination

Equality

Consistency

- System FC with Explicit Kind Equality, 2013
- Dependent Types in Haskell, draft
- Type Inference, Haskell and Dependent Types, 2013

Stephanie Weirich

Richard A. Eisenberg

Per Martin-Löf, Intuitionistic type theory (1984)

FORM∀L M∃THODS