Курс: Функциональное программирование
Практика 13. Алгоритм вывода типа

Типы данных

▶ В реализации алгоритма вывода типов для просто типизированного лямбда-исчисления (STT) мы будем использовать следующие типы данных:

```
infixl 4 :@
infixr 3 :->

type Symb = String

-- Терм
data Expr =
  Var Symb
  | Expr :@ Expr
  | Lam Symb Expr
     deriving (Eq,Show)

-- Тип
data Type =
  TVar Symb
  | Type :-> Type
    deriving (Eq,Show)

-- Контекст
newtype Env = Env [(Symb,Type)]
  deriving (Eq,Show)

-- Подстановка
newtype SubsTy = SubsTy [(Symb, Type)]
  deriving (Eq,Show)
```

Хотя подстановка типов вместо переменных типа `SubsTy` структурно устроена так же как контекст, но переменные в ней типовые, а не термовые.

Контексты, свободные термовые и типовые переменные

▶ (1 балл) Реализуйте следующий набор вспомогательных функций. Функция `freeVars` возвращает список свободных переменных терма

```
freeVars :: Expr -> [Symb]
freeVars = undefined
```

Функция `freeTVars` возвращает список свободных переменных типа (в STT все переменные типа свободные)

```
freeTVars :: Type -> [Symb]
freeTVars = undefined
```

Функция `extendEnv` расширяет контекст переменной с заданным типом

```
extendEnv :: Env -> Symb -> Type -> Env
extendEnv = undefined
```

Функция `freeTVarsEnv` возвращает список свободных типовых переменных контекста

```
freeTVarsEnv :: Env -> [Symb]
freeTVarsEnv = undefined
```

▶ (1 балл) Реализуйте функцию `appEnv`, позволяющую использовать контекст как частичную функцию из переменных в типы:

```
appEnv :: MonadError String m => Env -> Symb -> m Type
appEnv (Env xs) v = undefined
```

Обратите внимание на механизм обработки исключений, который мы будем использовать и в дальнейшем. Информация об исключении передается в виде строки; конкретная монада-обработчик исключений не фиксируется. Ожидаемое поведение:

```
GHCi> tx = (TVar "a" :-> TVar "b") :-> TVar "c"
GHCi> ty = TVar "a" :-> TVar "b"
GHCi> env = Env [("y",ty),("x",tx)]
GHCi> let Right res = appEnv env "x" in res
```

```
(TVar "a" :-> TVar "b") :-> TVar "c"
GHCi> let Right res = appEnv env "y" in res
TVar "a" :-> TVar "b"
GHCi> let Left res = appEnv env "z" in res
"There is no variable \"z\" in the enviroment."
```

Подстановка типа

▶ (1 балл) Реализуйте функции, осуществляющие подстановку типов вместо переменных типа в тип (`appSubsTy`) и подстановку типов вместо переменных типа в контекст (`appSubsEnv`):

```
appSubsTy :: SubsTy -> Type -> Type
appSubsTy = undefined

appSubsEnv :: SubsTy -> Env -> Env
appSubsEnv = undefined
```

▶ (2 балла) Реализуйте функцию, выполняющую композицию двух подстановок (носитель композиции является объединением носителей двух этих подстановок):

```
composeSubsTy :: SubsTy -> SubsTy -> SubsTy
composeSubsTy = undefined
```

Используя эту функцию сделайте тип `SubsTy` представителем класса типов `Monoid`.

Унификация и поиск главного типа

▶ (2 балла) Реализуйте алгоритм унификации, возвращающий для двух переданных типов наиболее общий унификатор или сообщение об ошибке, если унификация невозможна.

```
unify :: MonadError String m => Type -> Type -> m SubsTy
unify = undefined
```

Ожидаемое поведение:

```
GHCi> let Right sbs = unify (TVar "a" :-> TVar "b") (TVar "c" :-> TVar "d") in sbs
SubsTy [("a",TVar "c"),("b",TVar "d")]
GHCi> let Left tst = unify (TVar "a") (TVar "a" :-> TVar "a") in tst
"Can't unify (TVar \"a\") with (TVar \"a\" :-> TVar \"a\")!"
```

▶ (5 баллов) Реализуйте алгоритм построения системы уравнений на типы для заданных контекста, терма и инициализирующего типа для терма:

```
equations :: MonadError String m => Env -> Expr -> Type -> m [(Type,Type)]
equations = undefined
```

Возможное поведение:

```
GHCi> term = Lam "y" $ Var "x"
GHCi> env = Env [("x",TVar "a" :-> TVar "b")]
GHCi> let Right eqs = equations env term (TVar "o") in eqs
[(TVar "d",TVar "a" :-> TVar "b"),(TVar "c" :-> TVar "d",TVar "o")]
GHCi> let Left err = equations (Env []) term (TVar "o") in err
"There is no variable \"x\" in the enviroment."
```

Используя `equations`, реализуйте алгоритм поиска главной пары для терма бестипового лямбда-исчисления:

```
principlePair :: MonadError String m =>  Expr -> m (Env,Type)
principlePair = undefined
```

Возможное поведение (с точностью до имен типовых переменных):

```
GHCi> let Right pp = principlePair (Var "x") in pp
(Env [("x",TVar "a")],TVar "a")
GHCi> let Right pp = principlePair (Var "f" :@ Var "x") in pp
(Env [("f",TVar "a" :-> TVar "b"),("x",TVar "a")],TVar "b")
GHCi> let Right pp = principlePair (Lam "x" $ Lam "y" $ Var "y") in pp
(Env [],TVar "a" :-> (TVar "b" :-> TVar "b"))
GHCi> let Left err = principlePair (Var "x" :@ Var "x") in err
"Can't unify (TVar \"a\") with (TVar \"a\" :-> TVar \"b\")!"
```