Idris

2014 г.

Idris

- Haskellоподобный,
- с зависимыми типами,
- с опциональной проверкой на тотальность,
- строгий по-умолчанию,
- с тактиками,
- ...

Haskellоподобный

```
data MyList a = Nil | (::) a (MyList a)

(++) : MyList a -> MyList a -> MyList a
Nil ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

instance Functor MyList where
   map f Nil = Nil
   map f (x :: xs) = f x :: map f xs
```

Haskellоподобный

```
data MyList a = Nil | (::) a (MyList a)
```

```
instance Applicative MyList where
   pure x = [x]
   [] <$> _ = []
   (f :: fs) <$> xs = map f xs ++ (fs <$> xs)

instance Monad MyList where
   [] >>= _ = []
   (x :: xs) >>= f = f x ++ (xs >>= f)

test : MyList Int
test = do
   f <- [id, (*2)]
   x <- [3, 4]
   return $ f x
```

С зависимыми типами

```
data MyVect : Nat -> (a : Type) -> Type where
   Nil : MyVect 0 a
   (::) : a -> MyVect n a -> MyVect (S n) a

(++) : MyVect n a -> MyVect m a -> MyVect (n + m) a
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

infix 9 !!

(!!) : MyVect n a -> Fin n -> a
(x :: xs) !! fZ = x
(x :: xs) !! (fS y) = xs !! y
```

С опциональной проверкой на тотальность

```
total myHead : List a -> a
myHead (x :: xs) = x
```

```
Main.myHead is not total as there are missing cases
```

```
%default total
go : Int
go = go
```

```
Main.go is possibly not total due to recursive path
Main.go
```

В интерпретаторе:

`> :total f`

В типах вычисляются только тотальные функции.

Строгий по-умолчанию

```
broken : Int -> Int
broken 0 = 1
broken n = n * broken (n - 1)

ifThenElse : Bool -> a -> a -> a
ifThenElse True t _ = t
ifThenElse False _ f = f
```

`> ifThenElse True 0 (broken (-1))`

Интерпретатор:

`0 : Int`

Скомпилированный код (с точностью до оптимизаций):

`segmentation fault ./a.out`

С тактиками

```
module tactics

lemma_applicative_identity : (vs : List a) -> (pure id
    <$> vs = vs)
lemma_applicative_identity [] = refl
lemma_applicative_identity (v :: vs) =
   let rec = lemma_applicative_identity vs
   in ?lemma_applicative_identity_rhs

---------- Proofs ----------

tactics.lemma_applicative_identity_rhs = proof
  intros
  rewrite rec
  trivial
```

...

- Именованные инстансы

```
instance [myord] Ord Int where
    ...
sort @{myord} [2, 1, 3]
```

- Idiom brackets (для аппликативных функторов)

```
f : Maybe Int -> Maybe Int -> Maybe Int
f x y = [| x + y |]
```

- !-нотация (для монад)

```
g : Maybe Bool -> Maybe a -> Maybe a -> Maybe a
g x t f = if !x then t else f
```

- records

```
record R : Type where
    MkR : (f1 : Int) -> (f2 : String) -> R
```

...

- Опциональная ленивость

```
data Lazy : Type -> Type where
   Delay : a -> Lazy a

Force : Lazy a -> a
```

- Изменяемый синтаксис

```
syntax "if" [test] "then" [t] "else" [e] =
    boolElim test (Delay t) (Delay e)
```

- Минимальный вывод типов в where
- Гетерогенное равенство

```
data (=) : a -> b -> Type where
   refl : x = x
```

- auto

```
myCast : {auto prf : x = y} -> x -> y
myCast {prf=refl} x = x
```

Effects вместо трансформеров

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

tree : Tree Int
tree = Node 3 (Leaf 0) (Node 4 (Node 5 (Leaf 1) (Leaf
    2)) (Leaf 6))

dfs : (a -> { [STATE b] } Eff ()) -> Tree a -> { [
    STDIO, STATE b] } Eff ()
dfs f (Leaf x) = do
  putStrLn "Encountered␣leaf"
  f x
dfs f (Node x y z) = do
  dfs f y
  f x
  dfs f z

main : IO ()
main = print !(run eff)
 where eff : { [STDIO, STATE Int] } Eff Int
       eff = do
         dfs (\x => update (+x)) tree
         get
```

Type providers

```
import Providers

%language TypeProviders

strToType : String -> Type
strToType "Int" = Int
strToType _ = String

fromFile : String -> IO (Provider Type)
fromFile fname = return $ Provide $ strToType $ trim
    !(readFile fname)

%provide (T : Type) with fromFile "config.h"

f : T
f = 42
```

config.h

```
Int
```

TODOs

- Proof automation
- More better termination checker
- More better editor support (goto definition, autocomplete, . . .)
- More bindings (incl. low-level C bindings)
- More backends (e.g. GHC)
- Bugfixing