Idris

2014 г.

Idris

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

Haskellоподобный

$$\textbf{data}\ \mathit{MyList}\ a\ =\ \mathit{Nil}\ |\ (::)\ a\ (\mathit{MyList}\ a)$$

$$(++) : \mathit{MyList}\ a \to \mathit{MyList}\ a \to \mathit{MyList}\ a$$
$$[]\ ++\ ys = ys$$
$$(x :: xs)\ ++\ ys = x :: (xs\ ++\ ys)$$

$$\textbf{instance}\ \mathit{Functor}\ \mathit{MyList}\ \textbf{where}$$
$$\quad \mathit{map}\ f\ \mathit{Nil} = \mathit{Nil}$$
$$\quad \mathit{map}\ f\ (x :: xs) = f\ x :: \mathit{map}\ f\ xs$$

Haskellоподобный

```
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

$$\textbf{data}\ MyVect : Nat \to (a : Type) \to Type\ \textbf{where}$$
$$\quad Nil : MyVect\ 0\ a$$
$$\quad (::) : a \to MyVect\ n\ a \to MyVect\ (S\ n)\ a$$

$$(++) : MyVect\ n\ a \to MyVect\ m\ a \to MyVect\ (n + m)\ a$$
$$[] ++ ys = ys$$
$$(x :: xs) ++ ys = x :: (xs ++ ys)$$

$$\textbf{infix}\ 9\ !!$$

$$(!!) : MyVect\ n\ a \to Fin\ n \to a$$
$$(x :: xs)\ !!\ fZ = x$$
$$(x :: xs)\ !!\ (fS\ y) = xs\ !!\ y$$

Строгий по-умолчанию

$$broken : Int \to Int$$
$$broken\ 0 = 1$$
$$broken\ n = n * broken\ (n - 1)$$

$$ifThenElse : Bool \to a \to a \to a$$
$$ifThenElse\ True\ t\ \_ = t$$
$$ifThenElse\ False\ \_\ f = f$$

$$> ifThenElse\ True\ 0\ (broken\ (-1))$$

Интерпретатор:

$$0 : Int$$

Скомпилированный код(с точностью до оптимизаций):

```
segmentation fault ./a.out
```

С 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

С тактиками

```
lemma_applicative_identity : (vs : MyList 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

lemma_applicative_identity_rhs = proof
  intro a, x, xs, rec
  rewrite rec
  trivial
```