EDSL相关杂记（2）

Felis sap...
函数式编程、编程语言、编程 话题的优秀回答者
已关注

本期专栏讲述Expression Problem的解决方法之一：Data types à la carte。它实现可组合数据类型的原理是将数据类型中“递归”的语义抽离，将其变为函子，利用函子的可组合性实现数据类型的可组合性。

类型层面的不动点组合子

首先，我们讨论怎样改造一种最简单的EDSL AST：单类型（每种eval操作总是返回同一类型），且不涉及多种数据类型的互递归。以浮点数算术表达式为例：

```
data Op = Add | Minus | Mult | Div
data Expr = Lit Double | App Op Expr Expr
```

可以认为声明ADT即对已有类型进行运算，生成新类型。我们需要处理的情况有：

- 积类型（Product Type）：新类型的值同时包含一系列子类型的值
- 和类型（Sum Type）：新类型的值包含一系列子类型其中一种的值
- 递归类型（Recursive Type）：新类型的值可能包含自身类型的值

非递归的情况下，实现二元或多元的积/和类型轻而易举，比如可以用Tuple代表积类型，用Either代表二元和类型。怎样实现递归类型？我们先从type-level转移到term-level，思考类似的问题：lambda calculus不允许为term命名，无法进行自调用，如何实现递归函数？答案是使用不动点组合子。

```
fac :: I
fac 0 =
```

```haskell
-- now fac' has no recursion
fac' :: (Int -> Int) -> (Int -> Int)
fac' f 0 = 1
fac' f n = n * f (n - 1)

-- thanks to lazy evaluation
fix :: (a -> a) -> a
fix f = f (fix f)

-- works like fac
fac'' :: Int -> Int
fac'' = fix fac'
```

（不妨在草稿纸上手动规约一下fac'' 3，观察fix是如何工作的）

在类型层面，我们可以做类似的事情：

```haskell
data ExprF e = LitF Double | AppF Op e e

-- now we can tell expression tree height through type

e0 :: ExprF e
e0 = LitF 233

e1 :: ExprF (ExprF e)
e1 = AppF Add e0 e0

e2 :: ExprF (ExprF (ExprF e))
e2 = AppF Div e1 e1

-- .. and so on.

newtype Fix f = Fix { unFix :: f (Fix f) }

type Expr' = Fix ExprF

-- now we've recovered our recursive type
```

现在，我们将Expr的“递归”语义抽离，变成ExprF，然后通过运用类型层面的不动点组合子应用到ExprF上，恢复了递归类型。我们首先考虑新的Expr'定义如何使用：怎样写构造器和解释器。

```haskell
lit :: D
lit = Fi
```

```haskell
app op e0 e1 = Fix $ AppF op e0 e1

evalOp :: Op -> (Double -> Double -> Double)
-- definition omitted

evalExpr' :: Expr' -> Double
evalExpr' (Fix (LitF x)) = x
evalExpr' (Fix (AppF op e0 e1)) = evalOp op (evalExpr' e0) (evalExpr' e1)
```

看上去，除了多了一些语法噪音以外，构造器与解释器的实现与原来的Expr别无二致。这些额外的麻烦有何意义，与Expression Problem有何联系，很快就会揭晓。因为即将使用一系列高级的GHC扩展，我们需要一点准备工作：

```haskell
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE DeriveTraversable #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MagicHash #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NegativeLiterals #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

import Control.Monad
import Data.Functor
import GHC.Exts
```

可组合的函子

注意到一个关键的事实：ExprF是一个函子（Functor）！借助DeriveFunctor扩展，可以自动为ExprF生成Functor类实例。函子的一个好特性是可组合：一系列的函子可以组成函子的积/和类型，结果仍然是一个函子，可以实现fmap操作。并非所有代数结构都有这一好特性，比如单子（monad）的和就不一定满足单子性质。

回到正题。

```
data AppF e = AppF Op e e
    deriving (Functor, Foldable, Traversable)
```

现在，我们将LitF、AppF等EDSL的子集称为signature，每个signature的kind为* -> *，包含的这个类型标签e我们称之为hole。使用Data types à la carte实现EDSL的好处，就是在EDSL上的特定操作可分散给不同的signature在不同模块里加以实现。那么signature怎样组合成和类型，这种和类型怎样构造和解释?

原始文献里，函子的和/积类型均实现为二元操作，对多元组合则通过反复二元和/积实现，实现简单，所需GHC扩展也较少，但是有一定的缺点（需要有争议的OverlappingInstances扩展；二元组合需满足右结合的顺序，等等）。与原始文献中采用的二元和不同，这里我们实现一个多元和。前方高能代码!

```
data Sum :: [* -> *] -> * -> * where
    Here :: f a -> Sum (f ': fs) a
    There :: Sum fs a -> Sum (f ': fs) a

type family AllSatisfy (c :: k -> Constraint) (l :: [k]) :: Constraint where
    AllSatisfy _ '[] = ()
    AllSatisfy f (l ': ls) = (f l, AllSatisfy f ls)

deriving instance AllSatisfy Functor fs => Functor (Sum fs)
deriving instance AllSatisfy Foldable fs => Foldable (Sum fs)
deriving instance (AllSatisfy Functor fs, AllSatisfy Foldable fs, AllSatisfy Traversab

data N = Z | S N

type family FindElem (l :: [k]) (a :: k) :: N where
    FindElem (x ': _) x = 'Z
    FindElem (_ ': xs) x = 'S (FindElem xs x)

class HasElem' fs f (i :: N) where
    inj' :: Proxy# i -> f a -> Sum fs a
    prj' :: Proxy# i -> Sum fs a -> Maybe (f a)

instance HasElem' (f ': fs) f 'Z where
    inj' _ = Here
    prj' _ (Here v) = Just v

instance
    inj'
```

```haskell
class HasElem fs f where
    inj :: f a -> Sum fs a
    prj :: Sum fs a -> Maybe (f a)

instance (HasElem' fs f (FindElem fs f)) => HasElem fs f where
    inj = inj' (proxy# :: Proxy# (FindElem fs f))
    prj = prj' (proxy# :: Proxy# (FindElem fs f))

type f :<: fs = HasElem fs f
```

代码里Sum的实现细节无须理解，我们只需要记住其用法即可：

- Sum '[f, g, ..] a是f a, g a, ...等一系列值的nested Either类型，也就是多元的函子和
- 当f, g, ...等均为Functor实例，则Sum '[f, g, ..]为Functor实例；Foldable/Traversable亦然
- 当类型构造器f是类型构造器列表fs的成员时，则满足HasElem fs f，该类型类有inj（inject）、prj（project）方法，分别可以将f a注入到Sum fs a中，或尝试从Sum fs a中提取f a
- 因为HasElem会频繁用到，所以实现一个类型别名，f :<: fs即代表HasElem fs f

可组合的构造器

前面，我们定义了浮点数算术运算EDSL的两个signature，LitF和AppF；我们有了Sum这一工具可用于实现多个函子的和；我们可以用Fix将signature加入递归的语义，变为完整的表达式类型。现在，怎样组合这些工具，实现表达式的构造器？

```haskell
type Expr fs = Fix (Sum fs)

lit :: LitF :<: fs => Double -> Expr fs
lit = Fix . inj . LitF

app :: AppF :<: fs => Op -> Expr fs -> Expr fs -> Expr fs
app op e0 e1 = Fix (inj (AppF op e0 e1))

e :: (LitF :<: fs, AppF :<: fs) => Expr fs
e = app Add (lit 233) (lit 666)
```

我们实现了两个smart constructor，但它们处理的表达式类型是多态的！我们规定表达式类型Expr有一个类型标签fs，其kind为[* -> *]，即组成这种表达式的signature列表。现在，lit和app的类型签名也就很容易懂了：只要表达式包含LitF/AppF的支持，那么lit/app就能构造出该种表达式，而表达

于LitF/lit

下面看看怎样解释我们的可组合表达式。最粗暴的方法，可以写一个

```
evalExpr :: Expr '[LitF, AppF] -> Double
```

然后实现过程直接基于递归与模式匹配。但我们希望实现可组合的解释器，比如新增一个signature时，其他signature的解释器实现不需要动，不需要重编译。这是办得到的：

```
type Alg f a = f a -> a

class Eval f a where
    evalAlg :: Alg f a

type family EvalCap c fs a :: Constraint where
    EvalCap _ '[] _ = ()
    EvalCap c (x ': xs) a = (c x a, EvalCap c xs a)

instance EvalCap Eval fs a => Eval (Sum fs) a where
    evalAlg (Here v) = evalAlg v
    evalAlg (There vs) = evalAlg vs

cata :: Functor f => Alg f a -> Fix f -> a
cata f (Fix t) = f (fmap (cata f) t)

instance Eval LitF Double where
    evalAlg (LitF x) = x

instance Eval AppF Double where
    evalAlg (AppF op x y) = evalOp op x y where
        evalOp :: Op -> Double -> Double -- omitting implementation

evalExpr :: (AllSatisfy Functor fs, EvalCap Eval fs a) => Expr fs -> a
evalExpr = cata evalAlg

r :: Double
r = evalExpr (e :: Expr '[LitF, AppF])
```

从上到下解读一下这段代码的意思：

- 我们定义了一个类型别名Alg f a = f a -> a，Alg是algebra的意思。f代表一个signature，Alg f a是一个代表“折叠一层”的函数：假如signature的hole类型为a，那么这个函数将整个signature折叠为a。
- Eval f a

- 我们定义了一个叫cata的组合子。cata的参数是一个代表“折叠一层”的algebra和一个用Fix构建起来的递归AST，然后将这个algebra递归地应用到AST上，自底向上进行折叠，直到返回a。为了将algebra应用到AST内层的节点，需要signature满足Functor实例。
- 最终的解释函数是evalExpr，它处理的AST类型也是多态的：只要signature列表里每个成员都存在折叠到a的algebra，并且是Functor实例，那么evalExpr就能将整个AST折叠到a。
- 为LitF和AppF定义了“折叠到Double”的algebra以后，EDSL的解释器完成了，可以用evalExpr将前面的表达式e计算出结果。这里手动标了一下类型签名，因为e和evalExpr都是多态的，不限定e的signature的话，有可能存在额外signature不满足evalExpr的constraint。

现在，这个解释器的实现不仅满足了可组合性的要求，而且揭示了一个源自范畴论的recursion scheme（递归设计模式）——catamorphism，通用化的fold操作。

更复杂的解释器

我们再举2个更复杂的解释器例子：一个是EDSL的AST变换，一个是monadic解释器。

首先讲AST变换。我们引入一个新的signature：NegF，代表将某个表达式的值取反。然后我们需要实现一个（可组合的）函数，将含NegF的表达式变换到不含NegF的表达式。

```
newtype NegF e = NegF e
    deriving (Functor, Foldable, Traversable)

neg :: NegF :<: fs => Expr fs -> Expr fs
neg = Fix . inj . NegF

class Desugar f fs where
    desugarAlg :: Alg f (Expr fs)

instance EvalCap Desugar fs gs => Desugar (Sum fs) gs where
    desugarAlg (Here v) = desugarAlg v
    desugarAlg (There vs) = desugarAlg vs

instance LitF :<: fs => Desugar LitF fs where
    desugarAlg = Fix . inj

instance AppF :<: fs => Desugar AppF fs where
    desugarAlg = Fix . inj

instance (LitF :<: fs, AppF :<: fs) => Desugar NegF fs where
    desu
```

这里，我们定义了Desugar类型类代表去掉NegF的desugar algebra，Desugar f fs代表signature f能够Desugar到类型标签为fs的表达式。照例，我们定义了LitF和AppF的Desugar实例（不做任何变换），以及Sum的Desugar实例。对于NegF，我们将其变换为“乘以-1”的表达式。最后，将cata应用到这个desugar algebra，我们得到一个能够将NegF signature去除的desugar function。

一般而言，基于cata定义可组合解释器时，如果fold结果类型比较简单，可以直接复用前面的Eval类型类以及evalExpr；结果类型比较复杂（比如Desugar支持生成多态的新表达式），直接复用Eval或者实现newtype wrapper会比较麻烦，适合的做法就是为每个不同的解释器定义一个类型类，用于描述不同操作所对应的algebra。

最后，cata操作可以推广到生成monadic value的cataM，从而便于实现monadic的解释器处理副作用（I/O、错误处理等）：

```
type AlgM m f a = f a -> m a

cataM :: (Traversable f, Monad m) => AlgM m f a -> Fix f -> m a
cataM f (Fix t) = join (fmap f (mapM (cataM f) t))

class EvalM m f a where
    evalAlgM :: AlgM m f a

type family EvalCapM c m fs a :: Constraint where
    EvalCapM _ _ '[] _ = ()
    EvalCapM c m (x ': xs) a = (c m x a, EvalCapM c m xs a)

instance EvalCapM EvalM m fs a => EvalM m (Sum fs) a where
    evalAlgM (Here v) = evalAlgM v
    evalAlgM (There vs) = evalAlgM vs

instance EvalM IO LitF Double where
    evalAlgM (LitF x) = putStrLn "Lit" $> x

instance EvalM IO AppF Double where
    evalAlgM (AppF op x y) = putStrLn "App" $> evalOp op x y

evalExprM :: (AllSatisfy Functor fs, AllSatisfy Foldable fs, AllSatisfy Traversable fs
evalExprM = cataM evalAlgM

r' :: IO Double
r' = eva
```

我们初步展现了Data types à la carte：怎样将EDSL的AST分离成不同的signature，并将其组合起来，实现多态的构造器和基于cata的解释器。篇幅所限，这里展示的是最简单的情况：EDSL是单类型的，不涉及互递归，解释操作上下文不敏感。

进一步了解Data types à la carte，不妨阅读Wouter Swierstra在JFP上的原文，以及Compositional Data Types。后者进一步解决了context-sensitive、GADT与互递归支持，并且实现了除了cata以外的多种其他recursion schemes，对应的compdata库还带有一系列Template Haskell函数可以为用户自定义的signature自动生成smart constructor等boilerplate code。

关于recursion schemes的实现以及更多范畴论与函数式编程的联系，可以从recursion-schemes库，以及大名鼎鼎的Functional programming with bananas, lenses, envelopes and barbed wire一文看起。

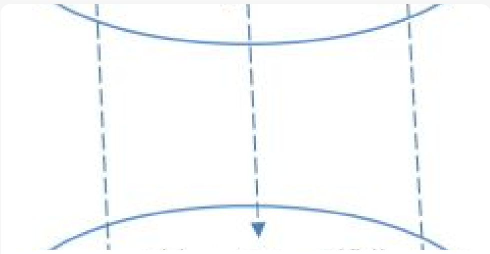
下期预告：下一期讲Finally Tagless style，作为另一种Expression Problem的解决方案。与本期的狂飙GHC特性相比，finally tagless常常使用普通的Haskell 2010特性即可使用，简单易懂，之后的文章里用到的频率也会更高。等到老后面讲Freer monad和effect system时，我们将回顾Data types à la carte并揭示它与finally tagless style的奇妙联系。

最后一点chit-chat时间：写专栏真是个不错的犯拖延症的方法啊。不过接下来一段时间要沉迷学习了，还有几个黄油待推，距离下次发布得等一段时间咯。。（

发布于 2016-10-10

Haskell 函数式编程 编程语言理论





EDSL相关杂记（1）
elis... 发表于雾雨魔法店

EDSL相关杂记：预告
Felis... 发表于雾雨魔法店

仙境里的Haskell（之四）——Functor类型类
大魔头-诺... 发表于魔鬼中的天...

仙境
大魔

14 条评论

切换为时间排序

写下你的评论...

Komoriii 1 年前

虽然看不太懂但是还是被grisia的图吸引进来了=_=

1

罗宸 1 年前

太给力了，，已经跟不上大大们的步伐了。。。这个代码写起来感觉好绕，不知道dependent type支持之后会不会好一点?

赞

Felis sapiens (作者) 回复 罗宸 1 年前

这篇并没用太多dependent type的特性。想看简单一些的代码可以看原始文章

赞 查看对话

罗宸 回复 Felis sapiens (作者) 1 年前

Data types à la carte 这个怎么翻译呢。。。google翻译是 “数据类型的地图”， 直接翻译 “à la carte” 结果是 “映射”

赞 查看对话

罗宸 1 年前

看起来，就是把每一种语法节点单独定义成一种Type，然后通过Type层面的Sum运算得到AST的定义，

Felis sapiens (作者) 回复 罗宸 1 年前

是的

赞 查看对话

Felis sapiens (作者) 回复 罗宸 1 年前

à la carte: (of a menu or restaurant) listing or serving food that can be ordered as separate items, rather than part of a set meal.

赞 查看对话

罗宸 回复 Felis sapiens (作者) 1 年前

谢谢，明白了，就是引申了“菜单”的意思啊。。

赞 查看对话

KurisuTiNa 1 年前

“lambda calculus不允许为term命名，无法进行自调用，如何实现递归函数？答案是使用不动点组合子。” ---但是 fix 本身不就是使用了 fix 这个名字的递归调用么?

赞

Felis sapiens (作者) 回复 KurisuTiNa 1 年前

你发现了亮点！不过不利用递归类型的话fix确实不能写出带类型的版本啦。。而且fix f = f (fix f)的定义也是最直观的（

2 查看对话

KurisuTiNa 回复 Felis sapiens (作者) 1 年前

这样啊...

赞 查看对话

方泽图 回复 罗宸 1 年前

偏向于“单点”（某一道菜）的意思。

换句话说，如果你的某个函数只需要Lit这一个限制（或者说证据），或者只需要Lit Add和Mul的组合，不需要别的，那么这个技巧就可以让你写出来的这个函数有这样准确的类型。同样，你也可以分开写多个简单的类型以及函数，然后再把它们组合起来。

想吃哪道菜就点哪道菜（添加一个新的 variant 不需要修改旧的代码），不需要点一个套餐然后引

1 查看对话

朱俸民 回复 Felis sapiens (作者) 1 年前

竟然还用法语。。。

赞 查看对话

neo lin 1 年前