♫ Free 地构造 eDSL ♫

荀洪道

二〇二四年六月四日

背景

又是一个重复入门的 Monad。2313

看过很多文章，它们的例子千篇一律用 Expr 说明，导致我一直不得要领。 Free Monad 确实擅于构造 AST（eDSL），但单纯的 Expr 对我来说没有实际作用，作为面向业务的码农，更喜欢贴近实际代码。 所以看了《Data types à la carte》和这篇《Introduction to Free Monads》， 总算知道 Free Monad 用来做什么，以及如何运用。

这篇只是简单总结一下使用方式，不会做一个复杂的 eDSL。 为了简单起见，我们只构建 sql 的事务提交：正常事务提交，以及正确地中断。

Free Monad

Free Monad 有什么作用？总结起来无非：

- 不用额外实现就能使用 do 记法
- 同一个 AST，在不同的上下文可以拥有不同的语义，有点类似于依赖注入

Free Monad 能做到这些，肯定封装了一些东西。接下我们去看看它的定义。

在此之前，我们看两个常见的例子，一个是列表（链表），另一个是二叉树：

```haskell
data List a = Nil | Cons a (List a)
data Tree a = Nil | Node a (Tree a) (Tree a)
```

他们至少有两个分支：一个 `Nil` 用于递归的停止；另一个 `Cons` 用于递归。 所以 Free 也有同样的结构：

```haskell
data Free f a = Pure a | Free (f (Free f a))
```

看起来有点抽象，尤其后半部分，`f (Free f a)`是一个递归定义，它要求 `f` 的 kind`* → *`，当然，更多情况下，我们需要 `f` 是一个 Functor。

构造 AST

之前讲了 `f` 是个 Functor，所以我们必须至少要有一个自由的类型变量，我们仅构造两个操作：

```haskell
data DatabaseF a = SelectF String (String → a) -- 执行 sql 语句
                 | AbortF -- 事务中断
    deriving (Functor)

type Database = Free DatabaseF -- 包装进 Free，享受 Free 带来的种种便利操作
```

SelectF 不能直接写 SelectF String a，原因在于 a 是操作的结果。试想一下这条语句：

```haskell
someAction = do
    result ← SelectF "select 1 + 1" ?
```

`?`该填什么呢？从数据库返回的数据固然都是 String，但这也要需要请求之后才能处理。 所以我们在此处填了`(String → a)`这个 continuation，String 表示数据返回的数据，它需要用户自行处理。 于是我们还需要额外处理 String 的类型类。

```haskell
class FromString a where
    decode :: String → a

instance FromString Int where
    decode = read

instance FromString String where
    decode = id
```

之后我们再为每个操作添加特定方法

```haskell
select :: FromString a ⇒ String → Database a
select sql = liftF $ SelectF sql decode

abort :: Database ()
abort = liftF AbortF
```

`select` 使用 `decode` 作为 continuation，这是很自然的，每次数据库查出来的数据，都要 decode 一遍。 `abort` 就很自然了。

解析 AST

我们先看下面两条语句:

```haskell
action = do
    a :: Int ← select "1 + 1"
    b :: Int ← select "1 + 2"
    pure $ a + b
```

```haskell
action = do
  a :: Int ← select "1 + 1"
  abort
  b :: Int ← select "1 + 1"
  pure $ a + b
```

它们要返回什么值呢？ 很明显，第一个返回的是 `a + b`， 但第二个语句就不一样，它用 `abort` 中断了整个事务，导致后面 `b` 取不出来，整个语句提前结束。 所以我们要让 AST 提前结束的可能。

嗯，除了 Cont，我们利用 MaybeT 也是能达到同样的效果。

```haskell
runAst :: DatabaseF a → MaybeT IO a
runAst (SelectF sql next) = do
    liftIO $ putStrLn sql
    pure $ next "10"
runAst AbortF = do
    liftIO $ putStrLn "ABORT"
    hoistMaybe Nothing
```

这里我们解构的是 DatabaseF，而不是 Database，这是因为我们可以用 `foldFree`，自动过滤掉 Free 外壳，直接操作我们的 AST， 最后我们再将整个 Free 拼回去:

```haskell
runDatabase :: FromString a ⇒ Database a → IO a
runDatabase ast = do
    putStrLn "BEGIN"
    r ← runMaybeT $ foldFree runAst ast
    case r of
        Just a → do
            putStrLn "COMMIT"
            pure a
        Nothing → error "failed"
```

运行看一下效果：

```haskell
runDatabase action
-- BEGIN
-- 1 + 1
-- 10 - 10
-- COMMIT
-- 20
```

输出 20，结果正确。

```haskell
runDatabase action
--   BEGIN
-- 1 + 1
-- ABORT
-- *** Exception: failed
-- CallStack (from HasCallStack):
--   error, called at /app/Main.hs:56:20
in main:Main
```

只输出一次 select，结果也符合预期。

结尾

从上面结果来看，我们确实构造出了数据库事务的 eDSL，虽然比较简单，但它已经能够自动补全“BEGIN”“COMMIT”等关键字；用户主动中断的情况下，也能提前退出当前事务。

我们再回过头来看 Free Monad，它首先要求我们自定义的类型必须是一个 Functor；当我们需要上下文的时候，一般需要用到 continuation 作为参数，例如 `SelectF` 不能直接填写 `a`，而是需要使用 `decode` 作为后续处理。从整个 eDSL 来看，除了我们的 DatabaseF 需要指定为 Functor 之外，并没有额外的要求，将 DatabaseF 包裹进 Free 之后，自然而然享受到 do 记法的便利。处理 AST 时，我们又可以结合 transformers，构造更加灵活的控制流，例如 MaybeT 的提前退出（理论上，ExceptT 更合适）。

希望这次入门，不再走歪了。