remote monad 设计模式

Felis sap...

函数式编程、编程语言、编程 话题的优秀回答者

已关注

从开始学编程起，我就思考过“是否能设计完美的编程语言”，而且显然并不只我一个人这样想过（有没有可能现在设计一个程序语言，它能够充分吸收现存语言的优点，同时排除那些缺点？ - 知乎）。对这个问题，不那么有趣的回答是：做不到，因为应用场景的多样性，blah blah blah……而我更喜欢另一种回答：做得到，只要面向不同的场景能够便捷地开发不同的 DSL 不就好了？

静态类型的函数式语言就非常适合各种 DSL （尤其是 Embedded DSL）的开发。（ref）在设计 DSL 时，需要处理的 issue 十分多样：表示 declarative/imperative 的计算；表示带 binding 和 scope 的计算；使用 host 语言的类型系统来保证 DSL 的类型安全；使 DSL 容易实现新的 primitive 和 interpreter，等等，而我们也有许多不同的设计模式来解决这些 issue。本文介绍 remote monad 设计模式，它可以用于 Haskell 中实现 monadic EDSL。

什么是 remote monad

在 Haskell 中，monad 类型类可以用于抽象描述 continuation-passing 的计算过程（以前的 Node.js 程序员应该很熟悉这种风格），应用极广，比如典型的 IO 类型就使用这个界面来编码带副作用的计算。

“什么是 monad” 这里略过不表，我们说说如何设计 monadic EDSL。如果 EDSL 需要表达 imperative 的计算，并且计算过程涉及到将中间值绑定到变量上，那么使用 monad 界面来表示，我们即可使用 Haskell 编译器为我们检查 DSL 的类型和作用域，有许多针对 monad 的组合子开箱即用，也可以使用 do-notation 像写普通的 imperative program 一样写 DSL 的程序。

DSL 的两大核心问题是“如何表示”和“如何解释”。对于 monadic EDSL，前一个问题的答案显而易见，至于后一个问题，我们有不同的答案：

签名中显式标注，最后在 IO monad （或者不涉及 IO 的话， Identity monad）中，提供这些计算过程所需的 context 并执行计算获得结果。
- 一部分计算过程在 Haskell 以外完成。解释执行时，Haskell 会通过 RPC（Remote Procedure Call）向外部系统发送指令并取得中间结果，而这些中间结果可以通过 Haskell 侧的函数处理并决定执行流。
- Haskell 测不执行任何计算，而是将整个 EDSL 程序序列化到另一门语言的程序，并输出编译后的程序，或将编译后的程序在其他平台上一次性运行后取得最终结果。

对于后两种解释方式，由于解释器涉及在 host 语言运行时以外的计算过程，所以相应 EDSL 的 monad 称之为 remote monad。

一个 monadic EDSL 的示例

我们先来设计一个 monadic EDSL。取名叫 Love monad 好了，让世界充满爱：

```
data Love :: * -> *

instance Functor Love
instance Applicative Love
instance Monad Love
```

Love 语言的“指令集”支持哪些操作呢?

```
queryCharacters :: Love [Person]
isMale, isFemale :: Person -> Love Bool
queryKoukando :: Person -> Love Double
confess :: Person -> String -> Love (Bool, String)
stab :: Person -> Love ()
```

这些指令在 Love monad 中执行以下操作：查询当前 route 的相关角色；查询指定角色的性别与好感度；向指定角色告白，并查询是否成功及其答复；用柴刀捅指定的角色。

Person 类型的定义我们暂且不用关心，暂且可以认为它代表当前攻略 context 中某个角色的 uuid。我们不能自己生成一个 Person 或者对其执行其他计算，只能在 Love monad 中通过给定指令进行操作。

不考虑 Love monad 的定义和解释器实现，我们已经可以写一些充满爱的程序片段了：

```
story :: Love ()
story = do
```

```
    isFuta <- (&&) <$> isMale character <*> isFemale character
    when isFuta $ do
        koukando <- queryKoukando character
        if koukando > 0.618
            then void $ confess character "好きです!"
            else when (koukando < 0) $ stab character
```

这段程序用到了前面的 Love 语言的指令集，以及一些 Haskell 自身的 monad 组合子。意思十分简单明了：查询并遍历角色列表（我们只关心其中的某类角色），查询角色好感度，如果好感度大于 0.618 就潇洒地告白（但并不关心告白的结果）；而如果好感度小于 0 就是仇敌，可以用柴刀了。

weak remote monad

接下来需要考虑 Love 语言的解释方式了。前面只规定了一个 interface，基于这个 interface 的程序有着许多可能的解释器实现方式，比如：

- 通过某个人生赢家的手机向其下达指令，手机上部署了训练好的模型可以识别角色、告白结果等并将结果传回服务器
- 通过调试器挂载到某个 galgame 的进程，监视其内存以收集结果，模拟键盘/鼠标事件以执行指令

等等。考虑到这是一篇以 Haskell 为主的教程，所以我们不考虑如何操纵一个人生赢家或者如何调试一个 galgame，而是通过另一个抽象的方式来编码 RPC 过程，然后只关心 Love 类型如何转换到 RPC 过程。

前面的 Love 程序中，涉及了两种不同的 monadic action：带返回值的，以及不带返回值的（即 Love () 类型）。在执行“带返回值的 action”时，我们需要通过 RPC 以 Haskell 值的形式获取结果，并在 Haskell 测进行相应计算后继续之后的 RPC。而对于不带返回值的 action，在 RPC 之后可以立即返回并继续计算。对应单次 RPC 的 monadic action 中，不带返回值的我们称之为 remote command，带返回值的则称为 remote call。

先考虑只有 remote command，没有 remote call 的情况。假设单次 RPC 的实质为“发送序列化的指令”：

```
newtype World = World { speak :: String -> IO () }
```

World 类型代表所有 RPC 过程的 context，包含一个发送单条指令的 callback。那么前面的 Love monad 实现也就呼之欲出了：

```
newtype Love a = Love { spreadLove :: ReaderT World IO a }
```

monad 中依次执行每条 remote command 所对应的 RPC。

当然，remote call 可以通过同一个思路实现：World 的定义变为“发送序列化的指令，接收序列化的结果”：

```
newtype World = World { speak :: String -> IO String }
```

而在实现一条 remote call 时，先通过 World 中的 callback 获取序列化的结果，再调用相应类型的 parser 解析结果并返回，解析失败时可以抛出 IO exception。

现在，Love 语言有了第一个解释器。这个解释器中，Love monad 采用的是 shallow-embedding，在拓展 Love 的新指令时，不需要修改 Love 的定义，只需要保证远端支持新指令的解析和执行，通过统一的序列化 - RPC 接口即可实现新指令。

这种最简单的 remote monad 我们称之为 weak remote monad：每一条 remote command/call 的执行，都对应一次解释器的 RPC 调用。

strong remote monad

有时，我们比较肉疼单次 RPC 调用的开销，希望在解释执行的过程中将多次 remote command 打包到单次 RPC 调用，这类 remote monad 我们称之为 strong remote monad。

需要明确的一点是：能够打包执行的，是连续的一系列 remote command。remote call 不能打包执行，因为 remote call 的返回值需要在 Haskell 侧用 Haskell 函数进行处理，并且有可能修改之后的执行流。

将 weak remote monad 修改为 strong remote monad 分为 2 步。首先解释器的 context 类型（World）需要支持“发送一批 remote command”的操作：

```
data World = World
    { speak :: String -> IO String
    , batch :: [String] -> IO ()
    }
```

而 Love monad 的实现则需要增加“缓冲”的机制：

```
newtype Love a = Love { spreadLove :: StateT [String] (ReaderT World IO) a }
```

实现单条 remote command 时，将序列化的指令存入 State monad 管理的缓存列表；单条 remote call 则先调用 batch 发送缓存的所有 command，清空缓存，再执行普通的 remote call。整个解释器的入口传入空列表作为 State monad 的初始状态，执行完毕后检查缓存是否为空，非空则最后执行一次 batch。

前面的 weak/strong remote monad 中，Haskell 运行时分担了一部分计算工作：remote call 的结果以 Haskell 值的结果返回，可以用 Haskell 函数进行处理。有时，我们只用 monad 界面来作为管理 binding 的语法糖，并不使用其他的 Haskell 函数处理中间值，这时我们希望“所有计算过程都在外部系统中完成，一次 RPC 搞定”。之前我们的序列化只针对单条 remote command/call，现在我们需要将整个 monadic EDSL 程序进行序列化。

首先需要在序列化的结果中表示 binding，这类 binding 我们称之为 remote binding。我们需要维护一个全局状态，用于生成 fresh identifier：

```
type Name = Int

data LoveState = LoveState
    { freshName :: Name
    , programBuffer :: String
    }
```

而 Love monad 的解释模型也发生了改变：

```
newtype Love a = Love { spreadLove :: State LoveState a }
```

之前的 Love monad 执行 remote command/call 的方式是使用 World 中包含的 IO callback 进行 RPC 调用，现在则不进行 RPC 调用，而是随着执行过程，在 programBuffer 末尾不断写入新生成的、带有变量名的语句。

“将某次计算的结果绑定到一个变量”的实现方式也发生了改变。之前我们使用 Love monad 自身的 monadic bind 操作，将结果绑定到 Haskell 值，现在需要改成“显式地创造一个变量，并将某次计算赋值到该变量”：

```
newtype Var a = Var { unVar :: Name }
newtype Expr a = Expr { unExpr :: String }

newVar :: Love (Var a)
getVar :: Var a -> Expr a
putVar :: Var a -> Expr a -> Love ()
execExpr :: Expr a -> Love ()
```

Name 代表 codegen 流程中处理的 raw identifier，而 Var 则是带有一个 phantom type 的 Name 以保证类型安全。现在我们不再将所有的计算都直接编码到 Love 类型中，而是单独设计一个 Expr 类型代表目标语言的 AST Expression（不过这里没有树，而是直接拍扁了存 String 了事——反正从子表达式组装的时候会自动在两边加括号）。而一段 Love 程序可以做这几件事，其中会修改全局状态 LoveState：

- 将某个 Expr 的值赋予某变量
- 将某个 Expr 进行求值，但并不赋值

现在，虽然 Love 语言还披着 monad 的皮，但是表达式的计算过程可能带有（目标语言中的）副作用。我们前面的那几个 Love 指令（查询、告白、柴刀）的类型签名也变了，它们不会返回 Love 类型的值，而是返回 Expr，这些 Expr 可以通过 putVar 或 execExpr 组装到一个 Love 程序之中。最后，在解释一个 Love 程序时，传入初始状态，运行 State monad，最后 LoveState 中的 programBuffer 就是序列化完成的整段程序。

关于赋值与副作用：大致有 3 种风格的接口，都可以用以上方法实现：

- C-style，表达式求值结果分为 L-value 和 R-value，其中 L-value （包含变量）可以对其赋值；赋值操作本身即表达式；表达式求值过程带副作用
- ML-style，有单独的 ref 类型代表引用，可以通过 readRef 和 writeRef 进行引用读写。表达式求值过程仍然带副作用
- Haskell-style，不带副作用的 expression 和带副作用的 monadic action 彻底分开。照理说 host 语言是 Haskell，那么我应该在 codegen monad 里面搞个 guest monad 来管理目标语言的副作用嘛——拉倒吧，你都生成字符串了还谁管安全不安全（逃

接口风格变了，局限性也就立即暴露出来了——首先，目标语言里面的表达式泄露不到 Haskell 侧也就不能用 Haskell 的函数去处理他了，所以要在目标语言里面做计算（算术计算、调用函数等），我们需要在 Haskell 这边手动实现这些计算的接口（看起来像是 sin :: Expr Double -> Expr Double 的玩意，实际上根本没算 sin，只是生成了对应的表达式）。

至于用 monad 组合子来操作控制流——比如 replicateM_ 1024 (execExpr (confess mk "Marisa-chan daisuki")) 仍然可以用，但是具体到生成代码时，会直接把"confess(mk, 'Marisa-chan daisuki');"序列化 1024 次。所以我们也不应依赖 monad 组合子来搞原来的循环之类功能了，而是实现自己的循环组合子，序列化到目标语言中的相应 loop statement。

另一个非常明显的局限性——在目标语言里使用 ADT 变得非常困难，事实上在我的阅读范围以内，所有完全序列化的 monadic EDSL 都只能操作简单的值（以及通过一些 trick 搞闭包），没有很 neat 的生成 ADT 的值并进行模式匹配的手段。我能想到的是用 Scott encoding 去强行模拟，通过 Template Haskell 偷到 ADT 的定义并生成相应的 boilerplate code，丑得惊天动地，所以有谁知道新进展或者能做到这一点的话，非常欢迎私信联系我，你来当一作。。

后记

Love 语言的底层指令和 Expr 类型的实现，为了省事，我们直接拍扁成 String 进行组合和传递，实际编程中，我们一般还是会用 ADT 来代表它们，以便编译优化。

对于需要完全序列化的 monadic EDSL，我们有另一种序列化方案，提供的 API 更接近“将中间值直接用 monad bind 提供”的风格，不需要显式新建变量，可以参考 Simple and

remote monad 的衍生工作：remote applicative functor，参考 Free delivery (functional pearl)，当我们不需要 monad 提供的 context-sensitive 计算能力时，使用 applicative functor 更有利于优化。

monadic EDSL 用于代码生成的又一个有趣的例子：An ASM Monad，实现了一个用于生成汇编代码的 monad。这个例子的意义在于不仅用到了 Monad 类型类，也用到了 MonadFix，用来实现 scope-safe 的标签和跳转。

remote monad 原始出处：The remote monad design pattern。文章末尾列举了许多 remote monad 概念提出之前就被独立发现并实现 DSL 的例子，作者自己也有实现有趣的 EDSL，包括用于操作 Arduino 和用于生成 JavaScript 的 EDSL。

Hackage 上实现了 monadic EDSL 的库数不胜数。我个人推荐学习的例子是 blank-canvas: HTML5 Canvas Graphics Library 和 sbv: Symbolic Haskell theorem prover using SMT solving.，分别用于生成 HTML5 Canvas 绘图代码和生成 SMT-LIB 代码调用 SMT solver。但是，滥用 monadic EDSL，只是因为 do-notation 看起来很好看的反面例子也是不少的，像测试框架和 web 框架就是重灾区，具体就不一一点名了。

总之，monad 什么的只是工具而已啦。关键是要向世界传达满满的 Love，不是吗（

编辑于 2017-04-22

Haskell 函数式编程

文章被以下专栏收录

Continuation 与 Monad

结构Monad 与 Continuation 都用于表示计算。用两个例子可以看到它们的结构非常相似：foo :: Maybe Int foo = Just 5 >>= \x -> Just 4 >>= \y -> return $ x ...

sdsjy

Monad in JavaScript

杨健 发表于前端黑魔法

Java(Vert.x)中的 Monad(Future)

圆胖肿

仙境 Mo

大魔

22 条评论

切换为时间排序

写下你的评论...

孙昌来 1 年前

恋妹！(

1

以上为精选评论

parker liu 1 年前

最近高產似母豬呀

赞

neo lin 1 年前

最近高產似母豬呀

赞

祖与占 1 年前

喜欢高产的司机!

2

李约瀚 1 年前

最近高產似母豬呀

赞

data LoveBlue :: * -> * -> * 不好吗（逃

赞

pangzi9 1 年前

智商受到10000点伤害

赞

parker liu 1 年前

两个小错误：
1. for_应该用forM_吧。
2. isFuta <- (&&) <$> isMale character <*> isFemale character中的(&&)应该是(||)吧。

赞

nameoverflow 1 年前

扶...futa?

赞

parker liu 1 年前

这就是论一般程序员学习编程语言设计的重要性，了解一些lambda演算的重要性。

赞

Felis sapiens (作者) 回复 parker liu 1 年前

1. for_ in Data.Foldable 2. 没错, 就是 && （逃

赞 查看对话

Belleve 回复 parker liu 1 年前

这是个梗，就要用 &&

赞 查看对话

杨玉印 1 年前

爽到◑▂◑

赞

祖与占 1 年前

> 但是，滥用 monadic EDSL，只是因为 do-notation 看起来很好看的反面例子也是不少的，像测试框架和 web 框架就是重灾区，具体就不一一点名了。 Hspec? 求点名啊 (

赞

嗯，hspec，还有几个用monad做routing的框架。。

赞 查看对话

 祖与占 回复 Felis sapiens (作者) 1 年前

好像 Scotty, Spock 这些轻量级的都用 Monad 吧 (

赞 查看对话

 罗宸 1 年前

“另一个非常明显的局限性——在目标语言里使用 ADT 变得非常困难” -- 这一段能举个例子么？因为感觉难或不难应该取决于目标语言是什么吧，比如说如果目标语言也是Haskell那应该也不难？

赞

 凉宫礼 1 年前

对啊 每次edsl都要造一遍加减乘除或非之类的轮子 好蛋疼

赞

 Felis sapiens (作者) 回复 罗宸 1 年前

这是针对目标语言非 Haskell、整个 monadic EDSL 程序都要序列化的情形

赞 查看对话

 parker liu 1 年前

在remote bind的形式中如何实现及时返回计算过程的中间结果给Host呢，比如我要计算一个耗时的循环，如果将循环中的单次结果返回给Host呢。

赞