使用 Haskell 解决逻辑谜题的编程实践

黄梓淳

Sep 20, 2025

逻辑谜题一直是考验人类推理能力的经典方式，例如著名的「骑士与骑士」谜题：在一个岛上，居民要么总是说真话（骑士），要么总是说假话（骑士），而你需要通过一系列陈述来推断真相。这类问题的核心在于定义约束条件，而不是描述具体的解决步骤，这正是声明式编程的用武之地。Haskell 作为一种纯函数式编程语言，以其独特的特性成为解决逻辑谜题的理想工具。Haskell 的纯函数性确保了推理过程的确定性，无副作用干扰；强大的类型系统帮助精确建模问题域，让非法状态无法表示；卓越的列表处理能力允许轻松生成和过滤解空间；而惰性求值则能高效处理甚至无限的可能性，只计算所需部分。通过 Haskell，我们可以像数学家一样定义问题，并让编译器自动找出答案。

1　热身：Haskell 武器库速览

在深入解决逻辑谜题之前，让我们快速回顾 Haskell 中一些关键工具。列表推导式是生成和过滤解的强大工具，其语法类似于数学中的集合表示法，例如 `[x | x ← [1..10], even x]` 会生成所有偶数，这让我们能够直观地表达解空间。Maybe 类型用于处理可能不存在的值，例如 `Just 5` 表示有值，而 `Nothing` 表示无值，这在检查约束时非常有用。Bool 类型表示真值，用于谓词函数。元组允许将多个值组合在一起，例如 `(1, hello)` 可以表示一个简单的实体。模式匹配则用于解构数据，例如在函数定义中匹配特定模式以执行不同逻辑。这些工具组合起来，为我们提供了声明式解决问题的基础。

2　实战演练：解构一个经典谜题

我们选择爱因斯坦逻辑谜题（又称「谁养鱼」谜题）作为案例，这是一个涉及多个属性（如国籍、颜色、饮料、宠物和香烟品牌）的复杂推理问题。解决过程分为几个步骤：首先，将谜题转化为规格说明，列出所有实体和线索；其次，用 Haskell 数据类型建立模型；然后，将约束条件编码为函数；最后，生成并筛选解空间。
第一步是定义数据类型来表示各种属性。我们使用代数数据类型来确保类型安全。例如：

```
data Nationality = Norwegian | Englishman | Swede | Dane | German deriving (Show, Eq)
data Color = Red | Green | White | Yellow | Blue deriving (Show, Eq)
data Drink = Tea | Coffee | Milk | Beer | Water deriving (Show, Eq)
data Pet = Dog | Birds | Cats | Horse | Fish deriving (Show, Eq)
data Cigarette = PallMall | Dunhill | Blend | BlueMaster | Prince deriving (Show, Eq)
```

这里，`deriving (Show, Eq)` 允许这些类型可显示和比较。每个数据类型代表谜题中的一个属性类别，确保我们只能使用有效的值。

接下来，我们定义表示一个房子的类型，通常使用元组或记录，但为了简洁，我们使用元组。一个房子由五个属性组成：国籍、颜色、饮料、宠物和香烟品牌。

```
type House = (Nationality, Color, Drink, Pet, Cigarette)
```

一个解决方案是五个房子的列表，代表一排房子。

```
type Solution = [House]
```

现在，我们需要编码所有线索作为约束函数。每个函数类型为 Solution → Bool，检查一个可能解是否满足条件。例如，第一条线索是「挪威人住在第一间房子」，我们可以这样写：

```
constraint1 :: Solution -> Bool
constraint1 ((Norwegian, _, _, _, _) : _) = True
constraint1 _ = False
```

这里，我们使用模式匹配来检查列表的第一个元素是否是挪威人。_ 是通配符，表示我们忽略其他属性。
另一条线索是「英国人住在红房子里」，我们需要检查整个列表中是否有这样一个房子。

```
constraint2 :: Solution -> Bool
constraint2 solution = any (\(nat, col, _, _, _) -> nat == Englishman && col == Red)
    ↪ solution
```

any 函数遍历列表，检查是否存在满足条件的元素。lambda 函数解构每个房子，比较国籍和颜色。
对于关系约束，如「绿房子在白房子的左边」，我们需要比较位置。

```
constraintLeftOf :: Solution -> Bool
constraintLeftOf houses = or [ greenIndex + 1 == whiteIndex | (greenIndex, (_, Green,
    ↪  _, _, _)) <- indexed, (whiteIndex, (_, White, _, _, _)) <- indexed ]
  where indexed = zip [0..] houses
```

这里，我们使用列表推导式和 zip 来获取索引，然后检查绿房子索引加一是否等于白房子索引。or 确保至少一对满足条件。
最后，我们生成所有可能解并应用约束。由于解空间巨大，我们利用列表推导式和惰性求值。

```
allHouses :: [House]
allHouses = [ (nat, col, drink, pet, cig) | nat <- [Norwegian, Englishman, Swede,
    ↪ Dane, German],
                                col <- [Red, Green, White, Yellow, Blue],
                                drink <- [Tea, Coffee, Milk, Beer, Water],
                                pet <- [Dog, Birds, Cats, Horse, Fish],
                                cig <- [PallMall, Dunhill, Blend, BlueMaster, Prince]
                                    ↪  ]

solutions :: [Solution]
```

```haskell
solutions = [ [h1, h2, h3, h4, h5] | h1 <- allHouses, h2 <- allHouses, h3 <-
    ↪ allHouses, h4 <- allHouses, h5 <- allHouses,
                            constraint1 [h1, h2, h3, h4, h5],
                            constraint2 [h1, h2, h3, h4, h5],
                            -- 应用所有其他约束
                            constraintLeftOf [h1, h2, h3, h4, h5] ]
```

这个列表推导式生成所有可能的房子排列，然后逐个应用约束函数过滤。由于 Haskell 的惰性求值，它只会在需要时计算，避免不必要的开销。

3 优化与思考：超越暴力破解

虽然上述暴力枚举方法正确，但效率低下，因为解空间随属性数量指数级增长。Haskell 提供了多种优化策略。首先，尽早过滤：在列表推导式中尽早应用约束，减少后续组合。例如，我们可以在生成房子时就应用部分约束，而不是生成所有后再过滤。其次，使用高级抽象如 State Monad 或 Logic Monad：库如 `logict` 提供智能回溯，类似于 Prolog 的搜索机制，能更高效地探索解空间。例如，Logic Monad 允许我们定义非确定性计算，并自动处理分支。第三，对称性剪枝：通过消除等价解（如颜色标签可互换）来减少搜索空间。尽管优化重要，但核心价值在于思维模式：我们声明问题是什么，而非如何解决，计算机负责繁琐搜索。这体现了函数式编程的优雅和抽象能力。

4 举一反三：其他谜题与模式

类似方法可应用于其他逻辑谜题。例如，数独问题可以用 Haskell 建模为网格，约束为行、列和子网格的数字唯一性；列表推导式可生成可能数字组合，并过滤无效解。八皇后问题则可通过生成皇后位置排列，并应用不攻击约束来解决。电路验证中，Haskell 的类型系统可确保连接正确，而约束函数检查逻辑一致性。共通模式是：定义问题域、生成解空间、施加约束、提取解。这种声明式方法不仅限于谜题，还适用于软件规范、测试用例生成和配置验证，展示 Haskell 在复杂问题解决中的通用性。

5 结论

通过 Haskell 解决逻辑谜题，我们展示了函数式编程的思维艺术：高度抽象、接近于问题描述的方式。这种方法强调定义而非执行，提升了代码的可读性和可维护性。价值 beyond 谜题，它提供了一种强大的问题解决框架，适用于多个领域。鼓励读者尝试用 Haskell 解决自己喜爱的谜题，或在社区中分享经验，进一步探索函数式编程的潜力。