函数式编程原理

课程报告

姓　　名：冯就康
学　　号：I201920029
班　　级：CS2002
指导教师：顾琳

计算机科学与技术学院

2023 年 10 月 16 日

Contents

一、函数式语言家族成员调研

1. 引言

函数式编程是一种编程范式，它将计算机程序的构建基础看作是数学中的函数，而非状态和可变对象。这种编程范式的主要目标是实现程序的逻辑明确，易于理解和修改。函数式编程语言家族中的成员众多，每一种都有其独特的特性和优点。

2. 函数式语言家族成员及其创始人

1. Lisp（John McCarthy）

Lisp 是由 John McCarthy 在 1958 年为 AI 研究而开发的，它是最早的函数式编程语言之一，对函数式编程的发展产生了深远影响。

Lisp 的主要特点是它的简洁性和表达力，它使用一种简单的语法，基于 S 表达式（符号表达式）来表示数据和代码。这使得 Lisp 具有强大的元编程能力，可以轻易地操作和生成代码，这在其他语言中往往很难实现。

Lisp 还引入了许多函数式编程的核心概念，如高阶函数和递归。高阶函数是指可以接受其他函数作为参数或返回函数的函数，这在Lisp中得到了广泛应用。递归是一种函数调用自身的编程技术，Lisp 通过尾递归优化支持了高效的递归。

Lisp 还创新了垃圾收集技术，这是一种自动管理内存的技术，它使得程序员可以专注于编程，而不需要手动管理内存，这对函数式编程的发展有着重要影响。

总的来说，Lisp 的设计哲学和技术创新对函数式编程产生了深远影响，许多现代函数式编程语言，如 Clojure 和 Scheme，都是直接或间接地受到了 Lisp 的影响。Lisp，一种由约翰·麦卡锡于 1958 年创立的高级编程语言，是最早的函数式编程语言之一，对计算机科学产生了深远影响。Lisp 最初的设计目标是为人工智能研究提供一种表达复杂数据结构的简单方式。在这种语言中，代码和数据都是由列表构成的，这一特性赋予了 Lisp 极高的灵活性和扩展性。作为一种“语言的语言”，Lisp 允许开发者使用 Lisp 书写运行时可以解释和执行的 Lisp 代码，这种能力使得 Lisp 在当今仍然被广泛用于计算机科学研究和大规模应用程序开发中。尽管 Lisp 从诞生以来经历过许多变化和扩展，如 Common Lisp、Scheme 和 Clojure，但它的核心思想和设计理念，如第一级函数、动态类型、递归、纯函数和垃圾回收等，仍深深影响着现代编程实践。

2. Haskell（Lennart Augustsson, Thomas Johnsson, Simon Peyton Jones 等）

Haskell 是一种用于教学、研究和工业应用的函数式编程语言。它具有类型类和单子输入/输出等特性，以逻辑学家 Haskell Curry 命名，其主要实现是格拉斯哥 Haskell 编译器。虽然 Haskell 在学术界和工业界有所应用，但并不广泛流行。它由一个委员会开发，经历了多个版本和更新，其中 Haskell 98 和 Haskell 2010 是其进化中的重要里程碑。Haskell 2010 引入了如分层模块名称等特性。

1987 年，由于函数式编程社区缺乏一种通用语言，一个委员会决定设计一种名为 Haskell 的语言。其主要目标是创建一种适合教学、研究和应用的语言，并且可以自由获取。后来，他们开发出了 Haskell 98 作为 Haskell 的稳定版本，并单独进行了一项标准化一组库的工作。1999 年，他们发布了 Haskell 98 语言和库报告。这些报告后来被修订，以纠正错误并进行改进。Haskell 继续在 Haskell 98 之外进行扩展。

Haskell 具有独特的特点，如语法、代数数据类型、类型系统、单子和输入/输出，以及对大规模编程的支持。它的多种实现和用于分析和调试的工具也得到了描述。此外，还讨论了使用 Haskell 构建的应用程序及其对各种用户社区的影响。这篇论文的重点是提供 Haskell 的历史，包括其起源和原则，技术贡献，实现和工具，以及应用和影响。然而，这篇论文承认，它无法涵盖 Haskell 所激发的所有贡献和创造力。

3. Erlang（Joe Armstrong）

Erlang 是一种并发式和功能性的编程语言和运行时系统，由 Ericsson 的 Joe Armstrong，Robert Virding 和 Mike Williams 在 1986 年发明，目的是改进电信系统的开发。Erlang 在功能性编程的发展历史上有着重要的地位。

在 1980 年代，Ericsson 希望创建一种适用于电信系统的编程语言，可以处理大规模并发操作，高可用性和热切换（无缝升级系统的能力）。为此，他们开发了 Erlang。Erlang 的设计原则是让系统易于更改和扩展，以适应不断变化的需求和环境。

函数式编程是一种编程范式，它将计算视为函数的求值，而不是改变变量的状态。Erlang 是一种纯函数式编程语言，这意味着它使用不可变的数据结构，没有副作用，并且函数调用没有副作用。

在函数式编程中，Erlang 提供了一些独特的特性，如模式匹配，尾递归优化和高阶函数。它的语法和其他函数式编程语言（如 Lisp 和 Haskell）有所不同，更接近于 Prolog。

Erlang 对函数式编程的发展产生了重要影响。它的并发模型，错误处理机制和热代码替换功能在其他许多语言中都找不到。

Erlang的成功引发了其他函数式并发编程语言的发展，如Elixir，它运行在 Erlang 的虚拟机上，带有 Erlang 的所有功能，并添加了更现代的语法和功能。

总的来说，Erlang 在函数式编程的发展历史中发挥了重要角色，帮助推动了并发和分布式系统的发展。

4. Scala（Martin Odersky）

Scala 是一种混合了面向对象编程和函数式编程的静态类型编程语言，由Martin Odersky在2003年开发。Odersky是Java泛型的共同设计者，他深受Java的影响，并希望将面向对象编程和函数式编程的优点结合在一起。

Scala的设计目标是解决 Java 的一些缺点，如冗长的语法和缺乏函数式编程能力。因此，Scala 支持更高级别的抽象，同时也提供了丰富的类型系统。

Scala 的一个独特特点是它的兼容性。Scala 能够与 Java 无缝交互，这使得它能够直接使用Java的类库，也能在Java虚拟机（JVM）上运行。这种互操作性使得Scala能够在现有的Java项目中逐步引入，而不需要一次性替换所有的代码。

Scala 在函数式编程中的影响主要体现在它如何将函数式编程的概念融入到一个主要面向对象的系统中。Scala 的函数是"一等公民"，可以像任何其他对象一样被传递和操作。此外，Scala 还支持模式匹配，尾递归优化，以及更复杂的函数式编程概念，比如Monad。

Scala 的成功在一定程度上激发了其他语言的发展，比如 Kotlin，它也是一种运行在 JVM 上的静态类型语言，结合了面向对象编程和函数式编程。总的来说，Scala 在函数式编程的发展历史中发挥了重要的作用，它的设计和实现为函数式编程的普及和接受打开了大门。

5. Clojure（Rich Hickey）

Clojure是一种现代的、动态的、功能强大的编程语言，由Rich Hickey在2007年创建。Clojure是基于JVM（Java虚拟机）的，因此它与Java有着良好的交互性。Clojure在设计上深受Lisp的影响，但也从函数式编程中借鉴了许多概念，例如不可变性和函数作为一等公民的概念。

Clojure 的主要特点是其强大的数据结构和对并发编程的支持。它的数据结构是不可变的，这意味着一旦一个数据结构被创建，它就不能被改变。这种不可变性使得并发编程变得更简单，因为它消除了许多并发编程中的复杂性。

Clojure 的另一个重要特点是其对并发编程的支持。并发是现代编程中的一大挑战，Clojure 通过提供一种叫做软件事务内存（STM）的机制来处理这个问题。STM 允许开发者在内存中执行事务，就像在数据库中执行事务一样，这极大地简化了并发编程。

Clojure 在函数式编程中的影响力可以从它对并发编程的处理和对不可变数据结构的使用中看出。这些特性使得 Clojure 在处理复杂的并发问题和大规模数据处理问题时，具有很大的优势。这也影响了其他函数式编程语言的发展，许多语言都开始引入并发编程和不可变数据结构的概念。

3. 各语言特性

1. Lisp：

Lisp 的主要特性是它的列表处理能力和宏系统。列表是 Lisp 的主要数据结构，而 Lisp 的宏系统则允许程序员创建新的语法结构，这让 Lisp 具有极高的灵活性和扩展性。

2. Haskell：

Haskell 的主要特性是它的纯函数式编程和强类型系统。在 Haskell 中，所有的函数都是纯函数，这意味着它们没有副作用。此外，Haskell 的类型系统非常强大，它可以检测出许多类型错误，从而使代码更加安全。

3. Erlang：

Erlang 的主要特性是它的并发模型和错误处理能力。Erlang 允许程序员创建大量的并发进程，并提供了强大的错误处理机制。

4. Scala：

Scala 的主要特性是它混合了面向对象和函数式编程。在 Scala 中，一切都是对象，同时也支持函数式编程的各种特性，如高阶函数、匿名函数等。

5. Clojure：

Clojure 的主要特性是它的强大的数据结构和对并发编程的支持。Clojure 提供了一套强大的、持久的数据结构，同时也提供了一套强大的并发编程工具。

4. 兴盛与没落的原因

函数式编程语言的兴盛主要源于下面几个原因：

- 并发和多核处理：函数式编程非常适合并发和多核处理，因为它不依赖于共享状态。在多核处理器的世界中，这是一个巨大的优势。
- 更易于测试和调试：由于函数式编程更注重纯函数（输出仅取决于输入的函数），因此更易于测试和调试。
- 代码简洁：函数式编程语言通常允许开发者用更少的代码和更高的抽象级别来解决问题。

然而，函数式编程也有其不足之处，导致其没落：

- 学习曲线：由于函数式编程的概念与传统的命令式编程非常不同，因此学习起来可能比较困难。
- 性能问题：尽管函数式编程适合并发和多核处理，但由于其不可变性和高度抽象，可能会产生性能问题。
- 缺乏主流接受度：尽管函数式编程语言如 Haskell 和 Erlang 等在某些领域（如并发和分布式系统）中受到欢迎，但在主流开发中，函数式编程的接受度仍然低于命令式编程语言如 Java、C++和 Python 等。

二、上机实验心得体会

实验二第五关：

本关任务：给定一个数组 A[1..n]，前缀和数组 PrefixSum[1..n]定义为：

PrefixSum[i] = A[0]+A[1]+...+A[i-1]；

例如：PrefixSum [ ] = [ ]

PrefixSum [5,4,2] = [5, 9, 11]

PrefixSum [5,6,7,8] = [5,11,18,26]

试编写：（提示：可借助其他函数来帮助完成功能）

(1) 函数 PrefixSum: int list -> int list，

要求：WPrefixSum(n) = O(n2)。(n 为输入 int list 的长度)

(2) 函数 fastPrefixSum: int list -> int list，

要求： WfastPrefixSum(n) =O(n).

解题思路：先分析一下 PrefixSum 函数和 fastPrefixSum 函数的主要逻辑，PrefixSum 函数是将一个元素的和作为新的元素添加到结果列表中。fastPrefixSum 则是对 PrefixSum 进行化函数，可以通过 PrefixSumHelp 辅助函数一次性计算出所有的前缀和，避免了重复计算的问题。

代码：

```
fun PrefixSum []=[]
    | PrefixSum [x]=[x]
    | PrefixSum (x::y::L)=x::PrefixSum(x+y::L);

fun PrefixSumHelp ([],a):int list=[a]
    | PrefixSumHelp (x::L,a)=a::PrefixSumHelp(L,a+x);

fun fastPrefixSum []=[]
    | fastPrefixSum (x::L)=PrefixSumHelp(L,x);
```

性能分析：PrefixSum 函数的时间复杂度是 O(n^2)，因为它需要对每个元素都进行求和操作。然而，fastPrefixSum 的时间复杂度优化到了 O(n)，因为它只需遍历列表一次。

遇到的问题：我在实现过程中遇到了一些困难，主要是在理解前缀和的计算过程和优化方法上。为了解决这个问题，我查阅了相关的教程和资料，通过实践和模拟计算过程，最终理解并实现了这两个函数。

心得和体会：在这个过程中，我深刻体会到了函数式编程的魅力，它的简洁性和强大的表达力让我印象深刻。同时，我也认识到了在解决实际问题时，需要不断地进行优化和改进，以提高程序的性能和效率。

实验四第三关：

编写函数 mapList，要求：
① 函数类型为: ((‘a -> ‘b) * ‘a list) -> ‘b list;
② 功能为实现整数集的数学变换(如翻倍、求平方或求阶乘)

编写函数 mapList2，要求：
① 函数类型为: (‘a -> ‘b) -> (‘a list -> ‘b list);
② 功能为实现整数集的数学变换(如翻倍、求平方或求阶乘)。

比较函数 mapList2 和 mapList，分析、体会它们有什么不同。

解题思路：先分析一下 mapList 函数和 mapList2 函数的主要逻辑，mapList 是一个高阶函数，它有一个函数和一个列表作为参数，然后应用该函数到列表的每个元素上[1]。它的类型为((‘a -> ‘b) * ‘a list) -> ‘b list，这意味着它可以接受任何类型的函数和列表，只要这个函数可以将列表中的元素类型转换为另一种类型。然后，mapList2 函数与 mapList 相似，但它的函数类型为(‘a -> ‘b) -> (‘a list -> ‘b list)。它首先接受一个函数作为参数，然后返回一个新的函数，这个新的函数接受一个列表作为参数，并应用初始函数到这个列表的元素上。

代码：

```
fun mapList(func,s) =
    case s of
        [] => []
        |(x::L) => (func x)::mapList(func,L);

fun mapList2(func) =
    let
        fun f[] = []
        |f(x::L) = func x::f(L)
    in f
    end
```

性能分析：由于这两个函数都是递归实现的，所以它们的时间复杂度都是O(n)，其中 n 是列表的长度。这意味着当列表的长度增加时，运行时间也会线性增加。

遇到的问题：在实现这两个函数时，主要的挑战是理解函数式编程的概念，特别是高阶函数。一旦理解了这些概念，编写函数就变得相对简单。我遇到的主要问题是如何正确地实现高阶函数，但通过阅读相关的资料和参考其他的 SML 代码，我最终能够成功地实现它。

心得和体会：这个任务让我更深入地理解了函数式编程的概念，特别是高阶函数和柯里化。我也学习了如何在 SML 中实现这些概念，以及如何分析函数的性能。我期待在未来的编程任务中继续使用和探索这些概念。

三、课程建议和意见

函数式编程原理课程主要介绍了函数式编程的基本概念、语法和实践应用。函数式编程是一种编程范式，它把计算视为函数而非命令的执行。在函数式编程中，函数是一等公民，可以作为参数传递，也可以作为返回值。这使得代码更加简洁，逻辑更加清晰。

课程建议：理论知识部分可以通过课堂讲解和阅读文献的方式进行学习，但实践应用部分则需要大量的编程练习。我认为实践是最好的学习方式。学生应该通过编写代码来理解和掌握函数式编程的概念。同时，我希望教学平台可以提供更多的实时反馈和代码评审，以帮助学生改进他们的编程技巧。

对于在线平台实验部署的改进意见：首先，平台应该提供更加详细的编程指南和教程，帮助初学者快速上手。其次，平台应该提更准确的编程反馈和错误提示，帮助学生及时发现和改正错误。另外，平台还应该设立一个讨论区，让学生可以交流学习经验和解决问题。

总的来说，对我个人来说，函数式编程原理这门课式一个既有趣又具有挑战性的一门课。作为一种编程范式，函数式编程以计算函数的评估为核心，这种方法让我看到了编程的另一面。这门课程内容深入而实质，让我对于如何构建和理解复杂的函数式程序有了更深的认识。同时，我也感受到了函数式编程的难度，有时候我需要花费大量的时间去理解和应用一些复杂的概念和技术。总的来说，我觉得这门课程是一次富有挑战性的学习经历，我期待在未来能够更好的掌握和应用函数式编程。