Introduction to Functional Programming

Alissa Tung

2 March 2024

This is an internal presentation version. Apologize for not listing all reference materials and contributions. I will mark all citations in future versions.

A Taste of Functional Programming

The History of Functional Programming Languages

The Curry-Howard Correspondence

Nix and NixOS

...from the famous Haskell.org Website

```haskell
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

...from the famous Haskell.org Website

由所有素数所构成的列表可以被表示为

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

1. 考虑全体大于或等于 2 的整数所构成的列表

...from the famous Haskell.org Website

由所有素数所构成的列表可以被表示为

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

1. 考虑全体大于或等于 2 的整数所构成的列表
2. 这一列表可以被解构为列表的头元素（此时是 2）和列表去掉头元素剩下的部分，分别记作 p 和 xs

...from the famous Haskell.org Website

由所有素数所构成的列表可以被表示为

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

1. 考虑全体大于或等于 2 的整数所构成的列表
2. 这一列表可以被解构为列表的头元素（此时是 2）和列表去掉头元素剩下的部分，分别记作 p 和 xs
3. 由于列表有序，从 xs 中筛去所有 p 的倍数所构成的新列表仍然有序。重复这一过程，每次得到的新的列表头元素都保证不是比它小的素数的倍数，并且是列表的最小元素。考虑素数的定义，它即是比 p 大的最小素数

...from the famous Haskell.org Website

由所有素数所构成的列表可以被表示为

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

1. 考虑全体大于或等于 2 的整数所构成的列表
2. 这一列表可以被解构为列表的头元素（此时是 2）和列表去掉头元素剩下的部分，分别记作 p 和 xs
3. 由于列表有序，从 xs 中筛去所有 p 的倍数所构成的新列表仍然有序。重复这一过程，每次得到的新的列表头元素都保证不是比它小的素数的倍数，并且是列表的最小元素。考虑素数的定义，它即是比 p 大的最小素数
4. 将每次列表解构出的列表头所组合而成的新列表即是由所有素数所构成的列表

...from the famous Haskell.org Website

```haskell
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

- 记号 `[2 ..]` 代表由所有大于或等于 2 的整数所构成的列表

...from the famous Haskell.org Website

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

- 记号 `[2 ..]` 代表由所有大于或等于 2 的整数所构成的列表
- 记号 `[x | x <- xs, x `mod` p /= 0]` 用类似集合论中 $\{x \mid x \in xs, x \bmod p \neq 0\}$ 的语法表示，从列表 `xs` 中筛去所有模 p 余数等于 0 的元素所构成的列表

...from the famous Haskell.org Website

```haskell
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

- 记号 `[2 ..]` 代表由所有大于或等于 2 的整数所构成的列表
- 记号 `[x | x <- xs, x `mod` p /= 0]` 用类似集合论中 $\{x \mid x \in xs, x \bmod p \neq 0\}$ 的语法表示，从列表 `xs` 中筛去所有模 p 余数等于 0 的元素所构成的列表
- 中缀二元算符 (:) 的左侧是一个整数，右侧是一个整数构成的列表；该算符用于解构或构造整数构成的列表

The Essence of Functional Programming

- Declarative
- Purely Functional
- Composable
- Higher-Order Functions
- Referential Transparency
- ...

The Essence of Functional Programming: Declarative

Haskell 允许程序员使用声明式的语言来

- 解决许多现实世界的编程问题
- 定义并计算数学上的概念

之所以能实现这一效果的关键是

- Haskell 支持使用递归的等式[1]来定义变元
- Haskell 是惰性求值的：一个变元只有在被需要时才会求值到具体的结果

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

[1] https://www.joachim-breitner.de/publications/rec-def-pearl.pdf

The Essence of Functional Programming: Purely Functional

在 Haskell 中，所有函数都是数学意义上的纯函数：

- 没有可变变元与赋值
- 没有副作用
- 没有循环、提前返回等命令式语言的常见控制流原语

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  (quicksort lesser) ++ [x] ++ (quicksort greater)
    where
      lesser  = filter (<  x) xs
      greater = filter (>= x) xs
```

Composable, Higher-Order Functions and Referential Transparency

考虑类型 A、B，由 A 元素所构成列表的类型记作 $[A]$。对于函数 $f\colon A \to B$，怎样将它从应用到一个类型为 A 的值 x，转换为应用到整个列表上？

Composable, Higher-Order Functions and Referential Transparency

考虑类型 A、B，由 A 元素所构成列表的类型记作 $[A]$。对于函数 $f : A \to B$，怎样将它从应用到一个类型为 A 的值 x，转换为应用到整个列表上？

```
fs :: [a] -> [b]
fs []       = []
fs (x : xs) = f x : fs xs
```

Composable, Higher-Order Functions and Referential Transparency

考虑类型 A、B，由 A 元素所构成列表的类型记作 $[A]$。对于函数 $f : A \to B$，怎样将它从应用到一个类型为 A 的值 x，转换为应用到整个列表上？

```
fs :: [a] -> [b]
fs []       = []
fs (x : xs) = f x : fs xs

g :: b -> c
gs :: [b] -> [c]
gs []       = []
gs (x : xs) = g x : gs xs
```

Composable, Higher-Order Functions and Referential Transparency

Higher-Order Functions：**此时可以将公共模式提取出**

```
map :: (a -> b) -> ([a] -> [b])
map f []       = []
map f (x : xs) = f x : map f xs

fs :: [a] -> [b]
fs = map f

gs :: [b] -> [c]
gs = map g
```

Composable, Higher-Order Functions and Referential Transparency

Composable：**可以通过函数组合构造新的函数**

```
hs :: [a] -> [c]
hs xs = gs (fs xs)
```

Composable, Higher-Order Functions and Referential Transparency

Composable：**可以通过函数组合构造新的函数**

```haskell
hs :: [a] -> [c]
-- hs xs = gs (fs xs)
hs = gs . fs
-- hs = map (g . f)

-- point-free style
foldl = flip . flip foldr id . (flip (.) .) . flip
```

Composable, Higher-Order Functions and Referential Transparency

Referential Transparency:

> *...One of the most useful properties of expressions is that called by Quine referential transparency. In essence this means that if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value. Any other features of the sub-expression, such as its internal structure, the number and nature of its components, the order in which they are evaluated or the colour of the ink in which they are written, are irrelevant to the value of the main expression.*

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- whenever you need a mutable variable to represent a mutable state, you may actually need a recursion

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- whenever you need a mutable variable to represent a mutable state, you may actually need a recursion
  - ```
    sum :: [Int] -> Int
    sum []       = 0
    sum (x : xs) = x + sum xs
    ```
    ```

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- whenever you need a mutable variable to represent a mutable state, you may actually need a recursion
  - ```
    sum :: [Int] -> Int
    sum []       = 0
    sum (x : xs) = x + sum xs
    ```
  - `gen_server` of Erlang/OTP
    ```

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- even side-effecting IO operations are but a description of what to do, produced by pure code

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- even side-effecting IO operations are but a description of what to do, produced by pure code
  - 类型系统确保了产生值的代码与产生带有副作用的函数不能错误地混用，指导用户使用产生副作用的函数来结合产生值的代码，从而构建新的描述副作用的函数

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- even side-effecting IO operations are but a description of what to do, produced by pure code
  - 类型系统确保了产生值的代码与产生带有副作用的函数不能错误地混用，指导用户使用产生副作用的函数来结合产生值的代码，从而构建新的描述副作用的函数
  - 由于这些函数也是纯函数式程序设计的一部分，纯函数式的一切优点仍然得到了保留：引用透明性、可组合等

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive without ruining its purity.

- even side-effecting IO operations are but a description of what to do, produced by pure code
  - 类型系统确保了产生值的代码与产生带有副作用的函数不能错误地混用，指导用户使用产生副作用的函数来结合产生值的代码，从而构建新的描述副作用的函数
  - 由于这些函数也是纯函数式程序设计的一部分，纯函数式的一切优点仍然得到了保留：引用透明性、可组合等
  - ```
    main = do
      putStrLn "Hello, what is your name?"
      name <- getLine
      putStrLn ("Hello, " <> name <> "!")
    ```
    ```

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with simple language builtin constructions.

- inductively define data types and recursively define functions is all you need

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with simple language builtin constructions.

- inductively define data types and recursively define functions is all you need
  - ```
    data Tm
      = Var Name
      | Lam Name Tm
      | App Tm Tm
      | Let Name Tm Tm
    ```
    ```

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with simple language builtin constructions.

- inductively define data types and recursively define functions is all you need
  - ```
    data Tm
      = Var Name
      | Lam Name Tm
      | App Tm Tm
      | Let Name Tm Tm
    ```
  - ```
    eval :: Env -> Tm -> Val
    eval env = \case
      Var x -> fromJust $ lookup x env
      App t u -> eval env t $$ eval env u
      Lam x t -> VLam x (\u -> eval ((x, u) : env) t)
      Let x t u -> eval ((x, eval env t) : env) u
    ```

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with beautiful abstractions.

- 在函数式程序设计的发展历史上，出现过许多优美的设计模式与实用的抽象框架，它们往往被称为 Functional Pearls

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with beautiful abstractions.

- 在函数式程序设计的发展历史上，出现过许多优美的设计模式与实用的抽象框架，它们往往被称为 Functional Pearls
  - Recursion Schemes 总结了常见的递归模式、怎样用简洁的方式来编写可靠且高效的递归程序、以及探究它们性质的方法

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with beautiful abstractions.

- 在函数式程序设计的发展历史上，出现过许多优美的设计模式与实用的抽象框架，它们往往被称为 Functional Pearls
  - Recursion Schemes 总结了常见的递归模式、怎样用简洁的方式来编写可靠且高效的递归程序、以及探究它们性质的方法
  - Free Monad 提供了解决表达式问题的一个思路，并且用类型系统与范畴论的性质指导了程序的编写，减轻了函数库作者和函数库用户的心智负担

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive with beautiful abstractions.

- 在函数式程序设计的发展历史上，出现过许多优美的设计模式与实用的抽象框架，它们往往被称为 Functional Pearls
  - Recursion Schemes 总结了常见的递归模式、怎样用简洁的方式来编写可靠且高效的递归程序、以及探究它们性质的方法
  - Free Monad 提供了解决表达式问题的一个思路，并且用类型系统与范畴论的性质指导了程序的编写，减轻了函数库作者和函数库用户的心智负担
  - Algebraic Effects 提供了对用户友好的系统化组织代码中副作用范围和粒度的方法，并且结合代数与形式语义方法确保了这一设计模式的正确与高效

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive while easier to write safer code.

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive while easier to write safer code.

- Free Theorems 和其他文献指出，类型系统与语言的形式语义都确保了程序一定程度上的正确性；这也是程序设计语言理论与类型论中常见的研究方法

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive while easier to write safer code.

- Free Theorems 和其他文献指出，类型系统与语言的形式语义都确保了程序一定程度上的正确性；这也是程序设计语言理论与类型论中常见的研究方法
- 在 Haskell 等函数式语言中，更容易编写完善且易用的测试框架、无锁数据结构与无锁程序等其他语言中容易出错的大型工程

Is Purely Functional Programming Languages as Expressive as Other Programming Languages

Yes, and even more expressive while easier to write safer code.

- Free Theorems 和其他文献指出，类型系统与语言的形式语义都确保了程序一定程度上的正确性；这也是程序设计语言理论与类型论中常见的研究方法
- 在 Haskell 等函数式语言中，更容易编写完善且易用的测试框架、无锁数据结构与无锁程序等其他语言中容易出错的大型工程
- 不同类型系统的变体与不同表达程度的类型驱动开发很大程度上减轻了函数式语言使用者的心智负担

Advanced Functional Programming Languages

- Statically Typed
- Type Inference
- Concurrent
- Modular
- Meta-Programming
- Proof-Carrying
- ...

The History of Functional Programming Languages

- Lisp: Recursive Functions of Symbolic Expressions and Their Computation by Machine (1960)

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
(deriv '(+ x 3) 'x)
1
(deriv '(* x y) 'x)
y
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

The History of Functional Programming Languages

- The ML Family: a family of general-purpose functional programming languages (1973)
  - ML
  - SML
  - Caml (Categorical Abstract Machine Language)
  - OCaml (Objective Caml)
  - Coq
  - ...
  - ```
    type ('a, 'e) telescope =
      | TNil of 'e
      | TCons of 'a * ('a, 'e) telescope Tm.bnd
    module Telescope (B : LocallyNameless.S) (E : LocallyNam
    sig
      include LocallyNameless.S with type t = (B.t, E.t) tel
      val bind : Name.t -> t -> t bnd
      val unbind_with : Tm.tm Tm.cmd -> t bnd -> t
      val unbind_all : Tm.tm Tm.cmd list -> t -> E.t
    end =
    struct
      type t = (B.t, E.t) telescope
    ```
    ```

The History of Functional Programming Languages

- The Miranda Family: a family of non-strict purely functional programming languages (1985)
  - Miranda
  - Haskell
  - Clean
  - Idris (strict)
  - Agda
  - ...
  - ```
    quicksort :: (Ord a) => [a] -> [a]
    quicksort [] = []
    quicksort (x : xs) =
      (quicksort lesser) ++ [x] ++ (quicksort greater)
        where
          lesser  = filter (<  x) xs
          greater = filter (>= x) xs
    ```
    ```

The History of Functional Programming Languages

- Erlang: Practical functional programming for a parallel world (1986)

```
even(Numbers) ->
    mapreduce(Numbers, fun(Number) -> Number rem 2 == 0
mapreduce(Numbers, Function) ->
    Parent = self(),
    [spawn(fun() -> Parent ! {Number, Function(Number)}
    lists:flatten(
        [
            receive
                {Number, true} -> Number;
                _ -> []
            end
         || Number <- Numbers
        ]
    ).
```

The Curry-Howard Correspondence

- propositions as types[2]
- proofs as programs
- simplification of proofs as evaluation of programs

[2] https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf

Dependent Sum Types

- given $A : \mathcal{U}$, and $B : A \to \mathcal{U}$, we can form a type $\Sigma(x : A), B(x)$
- $\Sigma(n : \mathbb{N}), x + 2 = 8$
- $(6, \mathrm{refl})$

Coproduct Types

- given $A : \mathcal{U}$ and $B : \mathcal{U}$, we can form a type $A + B$
- for $x : A$, constructor $\mathrm{inl} : A \to A + B$
- for $y : B$, constructor $\mathrm{inr} : B \to A + B$

The Curry-Howard Correspondence

- general mathematical theorem proving: dependent types - intuitionistic logic
- message-passing open programs following certain protocols: session types - linear logic
- ...

Nix and NixOS

Nix 是一个纯函数式、声明式的软件包管理器与构建工具。
Nix 语言是一门纯函数式、惰性求值的程序设计语言。
NixOS 是一个声明式的 Linux 发行版。
Nix 系列工具的目的是创建声明式的、可复现的构建和部署生态。

有关 Nix 和 NixOS 的所有概念都可以被视作为构造 AST 和运行解释器。

Nix Lang

在 Nix 语言中，所有对象都可以仅用两个概念表达：

- lambda
- attribute set

Nix Lang: lambda

lambda 是用于表示函数抽象的方式

```
f = x : x + 1
g = x : g : x + g
h = {x, y} : x + y
```

Nix Lang: attribute set

属性集是类似 JSON 的数据结构

```
{
  x = "hello";
  y = 123;
}
```

Nix Lang: Derivation

derivation 并不是某种特殊的语言机制，只是含有特定字段的属性集。
Nix 所构建和管理的所有软件包都是 derivation。
derivation 声明了一个软件包的

- 有关这个软件包的元信息
- 构建依赖（输入）
- 构建步骤
- 构建产物（输出）

除此之外，每个 derivation 还包含一些对这个 derivation 进行额外操作的辅助函数。
要研究 derivation，只需要研究它

- 是如何被创建的（构造 AST）
- 是如何被使用的（运行解释器）

Nix Lang: Derivation

```
$ nix derivation show /nix/store/z3hhlxbckx4g3n9sw91nnvlkjv
{
  "/nix/store/z3hhlxbckx4g3n9sw91nnvlkjvyw754p-myname.drv":
    "outputs": {
      "out": {
        "path": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswd
      }
    },
    "inputSrcs": [],
    "inputDrvs": {},
    "platform": "mysystem",
    "builder": "mybuilder",
    "args": [],
    "env": {
      "builder": "mybuilder",
      "name": "myname",
      "out": "/nix/store/40s0qmrfb45vlh6610rk29ym318dswdr-m
      "system": "mysystem"
```

Nix Lang: Derivation

要创建一个 derivation，必要的元素只有

- 有关这个软件包的基本元信息
- 构建依赖（输入）
- 构建步骤
- 构建产物（输出）

而额外的辅助函数和额外的元信息则是可以用函数自动计算出的。
但直接填写这一过程也是较为冗长的。Nix 提供了一些「返回值是 derivation」的函数，用于根据不同需求化简这一场景的复杂度。

Nix Lang: Derivation

常见的软件包往往形如

- 传统的 C++、GTK 项目：从 GitHub 获取源代码，使用 CMake 进行构建；需要用各种方法找到依赖
- Python 项目，使用依赖清单声明依赖，调用 `pip` 脚本进行构建
- Rust 项目，使用 `Cargo.toml` 做依赖管理，通过 `cargo build` 进行构建
- ...

Nix Lang: Derivation

针对这些场景，Nix 提供的「返回值是 derivation」的函数有

- `stdenv.mkDerivation`
- `buildRustPackage`
- `dockerTools.buildImage`
- ...

等许多 generic builder。

Nix Lang: Derivation

针对这些场景，Nix 提供的「返回值是 derivation」的函数有

- `stdenv.mkDerivation`，下面是使用的例子。一个 Nix 文件包含且仅包含一个 Nix 表达式，这些文件有时也被称作 Nix module。

```
{
  lib,
  stdenv,
  fetchzip,
}:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12.1";

  src = fetchzip {
    url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.t
    sha256 = "SHA256-somesha...";
  };
```

Nix Lang: Derivation

类似 stdenv.mkDerivation 的函数还可以接受一些参数作为声明软件包的依赖，如

```
nativeBuildInputs = with pkgs;
  [protobuf]
  ++ (
    if lib.strings.hasSuffix "-darwin" system
    then [darwin.apple_sdk.frameworks.SystemConfiguration]
    else [pkg-config]
  );

buildInputs = with pkgs; (
  if lib.strings.hasSuffix "-darwin" system
  then []
  else [openssl]
);
```

此处 pkgs 也是一个属性集，它有许多字段是 derivation。

Nix Lang: Derivation

构造完 derivation 后便可开始考虑如何使用一个 derivation，即用解释器去解释 derivation（AST）。

- 注意到一个 derivation module 实际上是一个 lambda，解释器会填上规约 lambda 所需要提供的参数
- `nix eval` 会将 derivation 计算到唯一确定的 hash 值，对应 Nix store 中的一个路径，
  形如
  `/nix/store/5p0ffp6bzac6bpjg30cjsw9dg7y6fxw4-ghc-9.4.8`
- `nix build` 会将 derivation 编译为具体的可执行的构建脚本，并执行构建操作
- 求出的 Nix store path 实际上和磁盘上的存储位置是一一对应的，通过控制 `installPhase` 可以手动控制 store path 中存储哪些内容；否则 Nix 的 builder function 会使用默认的存储 layout

Other Forms of Derivations: NixOS

如果一个 derivation 的 outputs（即存储在 Nix store path 的内容）是 Linux 系统的所有文件（`/etc`，`/bin`、所有软件），且有一个解释器负责配置并启动所有 `systemd` 服务、环境变量、符号链接等内容。我们就得到了一个声明式的 Linux 发行版，即 NixOS。

Other Forms of Derivations: Nix develop shell

如果一个 derivation 的 outputs（即存储在 Nix store path 的内容）是项目开发的所有依赖、环境变量、定制 shell 脚本等内容，且有一个解释器负责配置初始化脚本、环境变量、符号链接等内容。我们就得到了一个声明式的开发环境，并且带有所有需要的依赖。