函数式编程启示录

认知升级 = 效率 + 安全

刘洋 2020/09/12

作者其人

- 本科交通工程。因参加全国比赛接触了图像处理，从此入计算机坑
- 毕业后在家自学一年，与动物书为伴
- 几乎从来没有刷过题，leetcode 完成题数 < 200
- 第一次接触函数式编程：2012年：C++11，《黑客与画家》
- PLT (Programming Language Theory) 爱好者
- Haskell wiki 贡献者
- 如果赚够了400万美金，我就回老家读个 PLT PhD

提起函数式编程，你能想到什么？

函数式编程，不是（不只是）

- Immutability (and its concurrency-friendly characteristics)
- Laziness
- Purity
- Currying
- Totality
- Monad?!

大纲

- 类型 Types
- 全函数 Total Functions
- 并发 Concurrency
- 高级类型 Advanced Types

类型 Types

什么是类型?

- 它代表了数据的存储方式（representational）
- 它代表了所有可用的操作（behavioral）
- 它代表了可赋值的集合空间（*mathmatical*）
- 它代表了其指代的实际含义（semantical）

基数 Cardinality

他们有几个可能的取值?

bool

2

他们有几个可能的取值?

char

256

他们有几个可能的取值?

null

1

他们有几个可能的取值?

void

0

他们有几个可能的取值?

```
struct A {
  bool a;
  char b;
}
```

$$|A| = |\text{bool}| \cdot |\text{char}|$$

他们有几个可能的取值?

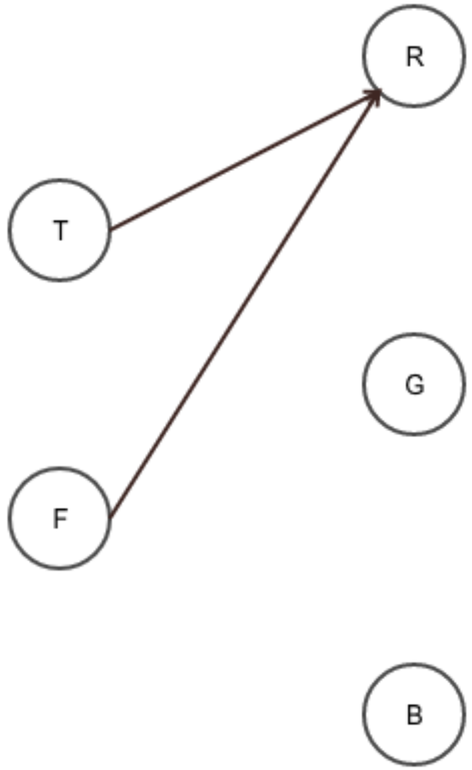
```
union B {
  bool a;
  char b;
}
```

$$|B| = |\text{bool}| + |\text{char}|$$

他们有几个可能的取值?

```
enum class Color { red, green, blue };
Color foo(bool);
```

他们有几个可能的取值?



他们有几个可能的取值?

```
enum class Color { red, green, blue };
Color foo(bool);
```

$$|\mathrm{foo}| = |\mathrm{Color}|^{|\mathrm{bool}|}$$

$$512 = 256 \cdot 2$$

struct { char; bool; }

积类型

Product Type

$$2 = 1 + 1$$

Bool = True | False

和类型

Sum Type

代数数据类型

Algebraic Data Type

- 当且仅当两个类型的基数相同，这两个类型等价
- 通过上述的加法/乘法/次幂操作，从旧类型中产生新类型

常见类型的代数表示

扩展阅读

List<T>

$$L(x) = 1 + xL(x)$$

$$L(x) = \frac{1}{1 - x}$$

$$L(x) = 1 + x + x^2 + x^3 + \ldots$$

常见类型的代数表示

扩展阅读

Tree<T>

$$T(x) = 1 + xT^2(x)$$

$$T(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

$$L(x) = 1 + x + 2x^2 + 5x^3 + \dots$$

常见类型的代数表示

扩展阅读

Set<T>

$$S(x) = 2^x$$

$$S(x) = x \to \text{bool}$$

学习这些理论有什么好处?

等价

等价意味着?

- 能相互替换使用而不损失功能
  - Color foo(bool) => bool foo2(bool) : 信息塌缩
  - int[5][6] => int[30]： 能相互替换，没有信息损失
- 作为函数输入时，具有相同的（良定义的）定义域

基数：找到类型的等价表达

基数：找到类型的等价表达

Set<T>

T -> Bool

基数：找到类型的等价表达

union { char; char; }

struct { bool; char; }

基数：发现类型与值空间的不匹配

基数：发现类型与值空间的不匹配

```
int a[4];
int b[5];
// a 与 b 类型相同吗？为什么？
```

$$|\text{int}|^4 \neq |\text{int}|^5$$

基数：发现类型与值空间的不匹配

在C语言中，int[4] 和 int[5] 是不同的两种类型

基数：发现类型与值空间的不匹配

```
// bad
int a[4];
a[3]; // 合法
a[4]; // 被翻译成了 *((int*)a + 4)，编译合法，但是逻辑错误

// 使用新的 array 类型的本质原因是让类型和值空间做匹配
std::array<int, 4> a;
a[3]; // 合法
a[4]; // std::array<int, 4> 不存在 [4] 操作，编译错误
```

和类型：让非法操作无法被表达

和类型：让非法操作无法被表达

- Java 带错误信息的返回类型
- TypeScript 带错误信息的返回类型
- PaymentMethod（积类型实现）
- PaymentMethod（和类型实现）
- 状态机（积类型实现）
- 状态机（和类型实现）

重新总结：什么是类型?

类型是数阶谓词逻辑的全称量词

$$\forall$$

类型是数据可用操作的全称量词

- 形式化验证 > 基于属性的测试 > 单元测试
- 实例1：求凸包算法
  - 算法 1
  - 算法 2
- 非函数式语言也能用 Property-based testing

类型的类型，类型的类型的类型，…

- 类型：Bool, Int, Char, …
- 类型的类型：
  - C++/Java: 泛型类型：List<T>, Map<K, V>
  - Haskel: 类型构造器：List a, Maybe a, Either a b
  - List：$x \to \frac{1}{1-x}$
- 类型的类型的类型
  - Haskell：类型构造器约束 Foldable = { List, Tree, Either a, … }
  - 假想：Listify：$(x \to \frac{1-\sqrt{1-4x}}{2x}) \to (x \to \frac{1}{1-x})$
- n阶类型：
  - Agda：Set_n
  - Agda: C++ 的泛型加法？太弱了！它只定义了值空间。我的泛型加法可以定义于任意阶类型空间！

类型 Q&A

全函数 Total Function

给下面的函数起名：

```
template <typename T>
T f(T);
```

identity

给下面的函数起名：

```
template <typename A, typename B>
A f(std::pair<A, B>);
```

first

给下面的函数起名：

```
template <typename T>
std::vector<T> f(std::vector<T>);
```

reverse, shuffle, …

给下面的函数起名：

```
template <typename T>
T f(std::vector<T>);
```

head, last

But it can be empty!

给下面的函数起名：

```
template <typename T>
std::optional<T> f(std::vector<T>);
```

Now it’s total

尽可能将函数写成全函数

Java’s null design makes it nearly impossible

尽量让所有的 if 都有匹配的 else

全函数 Q&A

并发，计算依赖

并发的代数含义?

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

$$a \oplus b = b \oplus a$$

结合律/交换律：顺序不重要

过程式范式：

```c
#include <stdio.h>

const int BUF_SIZE = 512;

int main(void) {
    char buff[BUF_SIZE];
    printf("your name: ");
    gets_s(buff, BUF_SIZE);
    printf("Hello, %s\n", buff);
    return 0;
}
```

声明式范式：

```
<!DOCTYPE html>
<html itemscope itemtype="http://schema.org/QAPage" class="html__responsive"
collamark="crx">
  ▶ <head>…</head>
... ▼ <body class="question-page unified-theme"> == $0
    <div id="notify-container"></div>
    <div id="custom-header"></div>
    ▶ <header class="top-bar js-top-bar top-bar__network _fixed">…</header>
    ▶ <script>…</script>
    ▶ <div class="container">…</div>
    ▶ <footer id="footer" class="site-footer js-footer" role="contentinfo">…
    </footer>
    ▶ <script>…</script>
    ▶ <noscript>…</noscript>
    ▶ <script>…</script>
    ▶ <div class="cm-popover" id="cm-popover" style="width: auto; left:
    685.211px; top: 2124.77px; display: none;">…</div>
    </body>
</html>
```

声明式范式：

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge [] bs = bs
merge as [] = as
merge (a:as) (b:bs) | a < b     = a : merge as (b:bs)
                    | otherwise = b : merge (a:as) bs

halve :: [a] -> ([a], [a])
halve [] = ([], [])
halve [x] = ([x], [])
halve (a:b:rs) = (a:as, b:bs)
  where (as, bs) = halve rs

mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort left) (mergesort right)
  where (left, right) = halve xs
```

如何尽量消除不必要的计算依赖?

如何尽量消除不必要的计算依赖

- 避免共享信息
- 避免副作用 Purity
- 不变性 Immutability
- 更简单的方式：将操作实现进满足结合律的结构中
  - List
  - Monoid
  - etc

Java 8 Stream

Java 8 Stream

```
List<Integer> listOfIntegers =
    new ArrayList<>(Arrays.asList(intArray));

Collections.sort(listOfIntegers, reversed);
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");
```

不同的默认行为有时会带来思维方式的革命

- 过程式：顺序 VS 声明式：非顺序 => 并发友好 => Google: MapReduce
- 线程：可换出 VS 协程：不可换出 => 资源友好 => Node.js
- C++：复制语义 VS Rust：移动语义 => 安全所有权 => 新的浪潮?

并发 Q&A

高级类型 Advanced Types

高级类型：断言逻辑正确性

```
// bonus: C++ 甚至能用类型断言函数的实现正确性（详见后续关于依赖类型的讨论）
template <typename T, size_t a, size_t b>
std::array<T, a + b> concat_array(std::array<T, a> u,
                                  std::array<T, b> v);
```

高级类型：断言逻辑正确性

- 依赖类型（Dependent Type）C++ 量纲分析示例
- 幽灵类型（Phantom Type）实现数据验证
- 课后作业：为 last4 实现幽灵类型 RegexString，满足 /\d{4}/ 约束

高级类型 Q&A

总结

- 识别非法定义域，选用窄化的更合适的类型
  - 用和类型拒绝非法表达式
- 尽量将函数实现为全函数
  - 让每个 if 都有匹配的 else
- CAAI: Concurrency As An Infrastructure
  - 实现并发逻辑 => 证明满足定律后装入并发容器
- 用高级类型实现细粒度的约束

更多话题

- 形式化验证 Formal Verification
- 单子（Monad）与可结合性（Composibility）
- …

参考与扩展阅读

- CppCon 2016: Ben Deane “Using Types Effectively"
- Haskell London: The Algebra of Algebraic Data Types
- Bartosz Milewski: Category Theory for Programmers

其他

- github.com/robturtle/io-interface TypeScript 运行时类型验证
- Learn Computer System in Python
- medium.com/@yl3710
- Side-project 集散地