类型和类型系统

冯新宇

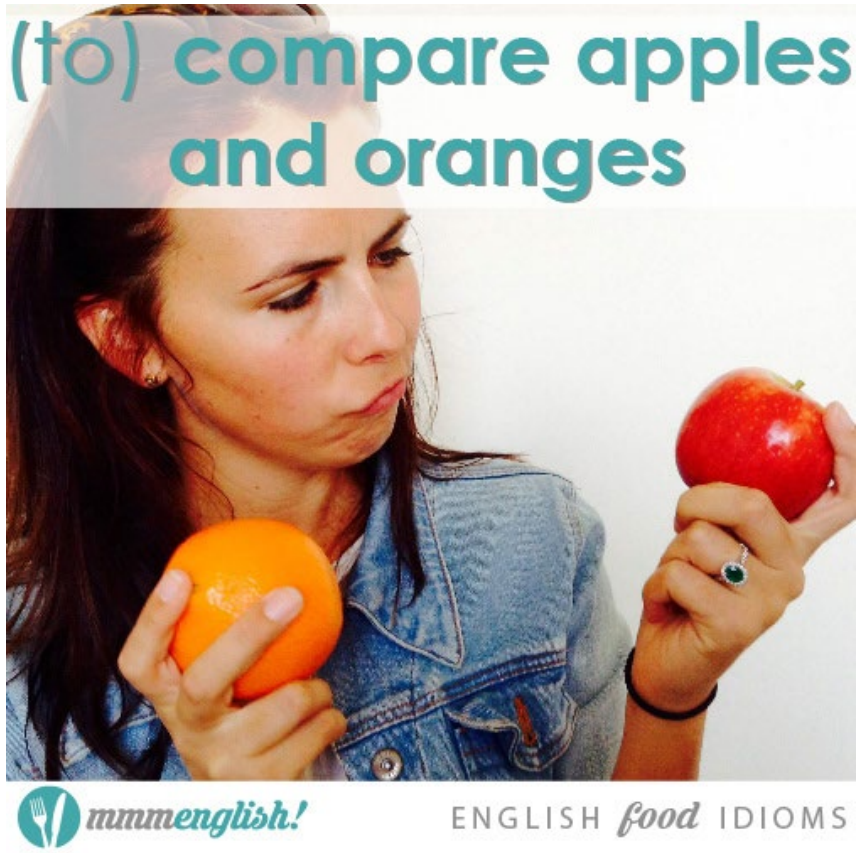
2025/09/17

为什么要类型系统?

类型的概念天然存在，无法回避

在编程中的体现：

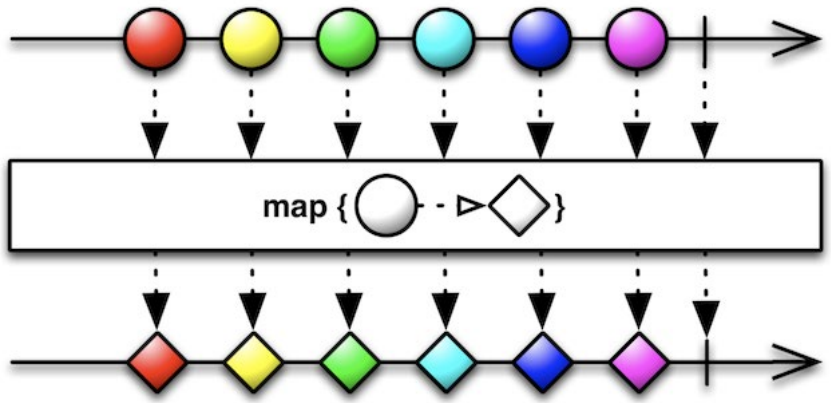
```
int* x, y, z;
int o = 4;
…
z = x + y; ✗
z = o + “123”;✗
```



类型系统在编程语言中的作用

- 提供一组约束，避免出错
  - 提升可靠性
  - 约束开发者的自由度
- 让程序更好读、更易维护
  - 类型信息让我们更容易预判程序行为
- 让编译器做更多优化
  - 更多的约束让开发者遵循更固定的模式
  - 类型给编译器提供了更多信息，指导优化

map: (A -> B) -> List<A> -> List<B>



动态类型语言

- 代表性语言：
  - Python, JavaScript, Lua, Ruby, PHP, Pearl, ...
- 变量的类型可以在运行中变化
  - 类型检查在运行中进行
- 解释执行为主
- 优点：
  - 约束少、表达能力强、易学易用
- 缺点
  - 性能差
  - 可靠性差
  - 难以支持大规模软件开发

静态类型语言

- 代表性语言：
  - Java, C#, Go, Swift, Scala, Rust ...
- 变量的类型在编译期确定，运行中不发生变化
  - 类型检查在编译期完成
- 虚拟机或静态编译
- 优点
  - 性能好
  - 可靠性好
  - 易维护，支持大规模软件开发
- 缺点
  - 约束多，有一定学习门槛

动态类型语言 vs. 静态类型语言

```
if (x > 0)
  y = “hello”;
else
  y = 3;

if (x > 0)
  z = y + “world”;
else
  z = y + 4;
```

动态类型：OK

静态类型：报错

假如你是会议组织者/主持者，要与参会人积极互动。。。

静态类型

动态类型

类型系统和类型检查

- 类型系统：一组类型定义和类型检查规则

  例：bool类型

  每种类型都有相应的Introduction规则和Elimination规则

Introduction rules

$$\frac{}{\vdash \text{true} : \text{bool}} \qquad \frac{}{\vdash \text{false} : \text{bool}} \qquad \frac{\vdash e1 : \text{bool} \qquad \vdash e1 : \text{bool}}{\vdash e1 \text{ and } e2 : \text{bool}}$$

Elimination rules

$$\frac{\vdash b : \text{bool} \qquad \vdash e1 : \text{T} \qquad \vdash e2 : \text{T}}{\vdash \text{if } b \text{ then } e1 \text{ else } e2 : \text{T}}$$

左右分支类型必须相同

类型检查规则

$$\frac{}{\vdash \text{true} : \text{bool}} \qquad \frac{}{\vdash \text{false} : \text{bool}}$$

$$\frac{\vdash e1 : \text{bool} \qquad \vdash e1 : \text{bool}}{\vdash e1 \text{ and } e2 : \text{bool}}$$

$$\frac{\vdash e : \text{bool} \quad \vdash e1 : \tau \quad \vdash e2 : \tau}{\vdash \text{if } e \text{ then } e1 \text{ else } e2 : \tau}$$

Judgment:

$\vdash e : \tau$

```
func check(e: Expr, T: Typ) : Bool
{   match(e){
        case true: return (T == bool)
        case false: return (T == bool)
        case e1 and e2:
            return check(e1, bool) && check(e2, bool)
        case if e then e1 else e2:
            return check(e, bool)
                    && check(e1, T)
                    && check(e2, T)
        ...
    }
}
```

类型的分类 —— 基本类型和复合类型

- 基本类型：整数、浮点数、布尔等

- 复合类型
  - 数组、struct、class等
  - 积类型、和类型

代数数据类型 —— 积类型

积类型：pairs and tuples

$$\frac{\vdash e1 : \tau1 \qquad \vdash e2 : \tau2}{\vdash (e1,\ e2) : \tau1 \times \tau2} \qquad \frac{\vdash e : \tau1 \times \tau2}{\vdash \boldsymbol{fst}\ e :\ \tau1} \qquad \frac{\vdash e : \tau1 \times \tau2}{\vdash \boldsymbol{snd}\ e :\ \tau2}$$

多元tuple可以看成是pair的泛化：

$(e1, e2, \dots, en)$ 可以看做是 $(e1, (e2, (\dots, en) \dots))$ 的简写，

其类型 $\tau1 \times (\tau2 \times (\dots \times \tau n) \dots)$ 可以表示为 $\tau1 \times \tau2 \times \cdots \times \tau n$

struct/records可以看做是带标签的tuple（即labelled tuples）：

$\{d1 : \tau1 = e1, d2 : \tau2 = e2, \dots, dn : \tau n = en\} : \mathit{struct\ MyStruct}\ \{d1 : \tau1, d2 : \tau2, \dots, dn : \tau1\}$

代数数据类型 —— 和类型

和类型：sum types and enum

(Types) $\tau ::= \cdots \mid \tau 1 + \tau 2$

(Terms) e ::= ⋯ | left e | right e | case e do e1 e2

Consider unions in C:

```
union data{
    int i;
    float f;
    char c;
}
```

Using the same location for multiple data.
Can contain only one value at any given time.

代数数据类型 —— 和类型

和类型：sum types and enum

(Types) $\tau ::= \cdots \mid \tau 1 + \tau 2$

(Terms) e ::= ⋯ | left e | right e | case e do e1 e2

$$\frac{\vdash \mathrm{e}: \tau 1}{\vdash \mathrm{left}_{\tau 1+\tau 2}\ e: \tau 1 + \tau 2}\ \text{(left)} \qquad \frac{\vdash \mathrm{e}: \tau 2}{\vdash \mathrm{right}_{\tau 1+\tau 2}\ e: \tau 1 + \tau 2}\ \text{(right)}$$

$$\frac{\vdash \mathrm{e}: \tau 1 + \tau 2 \qquad \vdash \mathrm{e1}: \tau 1 \to \tau \qquad \vdash \mathrm{e2}: \tau 2 \to \tau}{\vdash \mathrm{case\ e\ do\ e1\ e2}: \tau}\ \text{(case)}$$

代数数据类型 —— 和类型

和类型：sum types and enum

(Types) $\tau ::= \cdots \mid \tau 1 + \tau 2$

(Terms) e ::= ⋯ | left e | right e | case e do e1 e2

$$\frac{\vdash \mathrm{e}: \tau 1}{\vdash \mathrm{left}_{\tau 1+\tau 2}\ e: \tau 1 + \tau 2}\ \text{(left)} \qquad \frac{\vdash \mathrm{e}: \tau 2}{\vdash \mathrm{right}_{\tau 1+\tau 2}\ e: \tau 1 + \tau 2}\ \text{(right)}$$

$$\frac{\vdash \mathrm{e}: \tau 1 + \tau 2 \qquad \vdash \mathrm{e1}: \tau 1 \to \tau \qquad \vdash \mathrm{e2}: \tau 2 \to \tau}{\vdash \mathrm{case\ e\ do\ e1\ e2}: \tau}\ \text{(case)}$$

和类型在实际语言中的形式（仓颉）

```
enum Expr =
  | CONST(Int)
  | ADD(Expr, Expr)
  | MUL(Expr, Expr)
```

```
CONST(3)
ADD(CONST(3), COSNT(4))
MUL(CONST(3),
    ADD(CONST(4), COSNT(5))
)
```

```
func eval (e: Expr): Int {
    match(e) {
        case CONST(i) => i
        case ADD(e1, e2) => eval(e1) + eval(e2)
        case MUL(e1, e2) => eval(e1) * eval(e2)
    }
}
```

和类型在实际语言中的形式（仓颉）

```
enum Expr =
   | CONST(Int)
   | ADD(Expr, Expr)
   | MUL(Expr, Expr)
```

```
CONST(3)
ADD(CONST(3), COSNT(4))
MUL(CONST(3),
     ADD(CONST(4), COSNT(5))
)
```

对应于下面的和类型：

$$int + \big((expr \times expr) + (expr \times expr)\big)$$

和类型在实际语言中的形式（Swift）

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")

switch productBarcode {
   case .upc(let numberSystem, let manufacturer, let product, let check):
      print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
   case .qrCode(let productCode):
      print("QR code: \(productCode).")
 }
```

“积类型”和“和类型”

- “logical duals”
  - To make a $\tau 1 \times \tau 2$, we need a $\tau 1$ and a $\tau 2$
  - To make a $\tau 1 + \tau 2$, we need a $\tau 1$ or a $\tau 2$
  - Given a $\tau 1 \times \tau 2$, we can get a $\tau 1$ or a $\tau 2$ or both (our “choice”)
  - Given a $\tau 1 + \tau 2$, we must be prepared for either a $\tau 1$ or a $\tau 2$ (the value’s “choice”)
  - If $\tau 1$ has n1 possible values, and $\tau 2$ has n2 possible values,
    - $\tau 1 \times \tau 2$ has $n1 \times n2$ possible values
    - $\tau 1 + \tau 2$ has $n1 + n2$ possible values

“积类型”和“和类型”

- “unit”和“nothing”类型
- unit类型：只有一个值 “()”，可以视作特殊的tuple类型
  - T × unit ~ T
- nothing类型：没有值！可以视作0个分支的sum类型
  - T + nothing ~ T
- 各自有什么用?

函数类型

$$\frac{\Gamma, x : \tau 1 \vdash e : \tau 2}{\Gamma \vdash (\lambda x{:}\tau 1.\, e) : \tau 1 \to \tau 2}$$

$$\frac{\Gamma \vdash f{:}\tau 1 \to \tau \qquad \Gamma \vdash e{:}\tau 1}{\Gamma \vdash f(e) : \tau}$$

$$\frac{}{\Gamma,\ x : \tau \ \vdash\ x : \tau}$$

```
func check(Gamma: Env, e: Expr, T: Typ) : Bool
{
        …
}
```

Judgment:

$$\Gamma \ \vdash e{:}\tau$$

- Under Γ, e is a well-typed expr of type τ
- Typing context
  (records types of non-local variables)

$$\Gamma \ ::= \ \cdot \mid \ \Gamma, x{:}\tau$$

Type Checking vs Type Inference

- Standard type checking:

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Examine body of each function
  - Use declared types to check agreement

- Type inference:

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2; };
```

  - Examine code without type information. Infer the most general types that could have been declared.

经典的类型推断算法：
Hindley-Milner算法

类型推断

是否需要类型标注**不是**区分静态类型和动态类型的标准

```
let x = 100
let y = “hello”

print(x + 3)
print(y + “ world!”)
```

静态类型系统仍然允许省略类型标注

编译器可以自动推断类型信息

Hindley-Milner算法

参见Stanford CS242 (2012) 课件

Hindley-Milner算法

- 不足之处：
  - 参数化多态的表达力约束
  - 子类型的支持

类型的分类 —— 可变类型和不可变类型

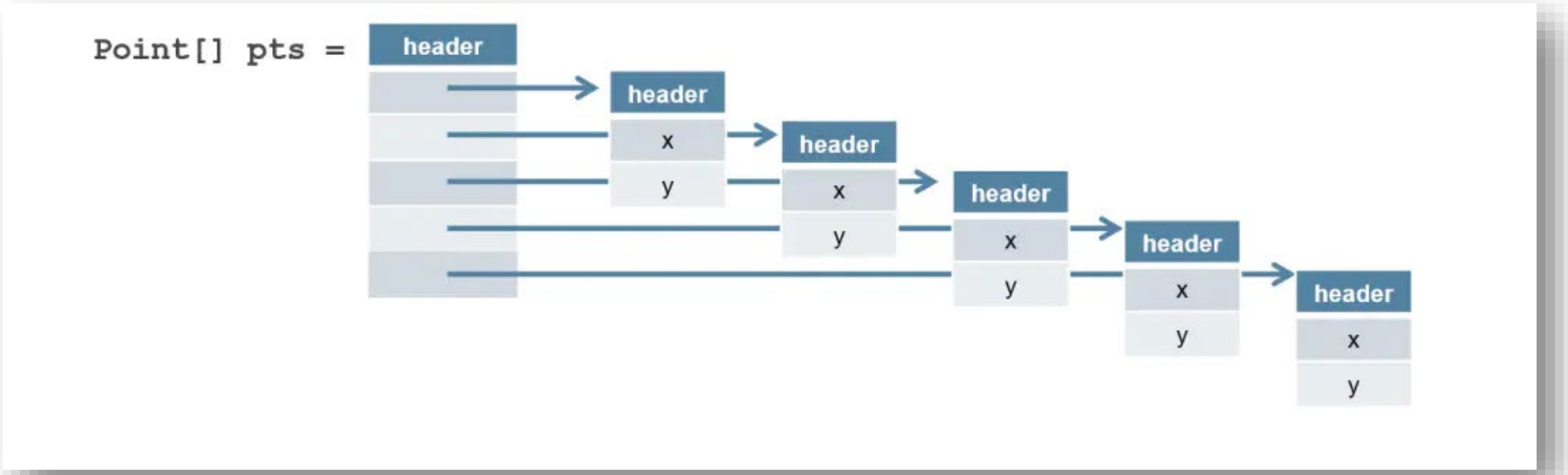
- 例：Python里面tuple和list的区别

类型的分类 ——值类型和引用类型

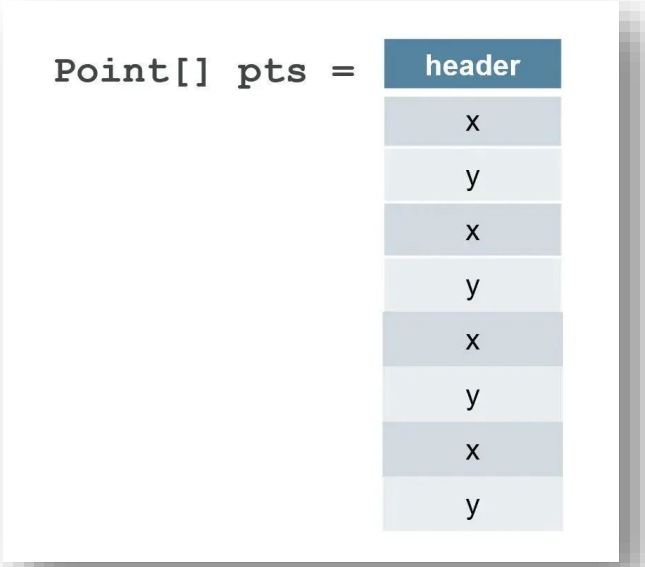
- 值类型
  - 类型所包含的值是数据本身（可以是复杂类型）
  - 赋值：copy语义，复制数据本身
  - 参考：C里面的struct

- 引用类型
  - 类型所包含的值本身是个引用，由引用间接的访问数据本身
  - 赋值：复制的是引用，构成了引用别名
  - 参考：Java里面的对象，Haskell里面的各种类型

引用类型



值类型



值类型优势

- 性能（速度和内存）
- 可靠性

https://www.baeldung.com/java-valhalla-project

值类型和引用类型的区别

- 语义上的区别
  - 赋值语义
    - 但对于不可变数据类型，我们感受不到上述区别
  - 不同的数据类型，表示上是否统一？
    - 会影响其他很多相关特性，接下来的内容中会介绍

- 性能开销上的区别
  - 成员访问的区别
    - 运行期进行间接访问还是编译期实现算好偏移
  - 内存占用和局部性的区别
  - 内存管理的区别
    - 栈上分配还是堆上动态分配