类型主导的编程语言设计

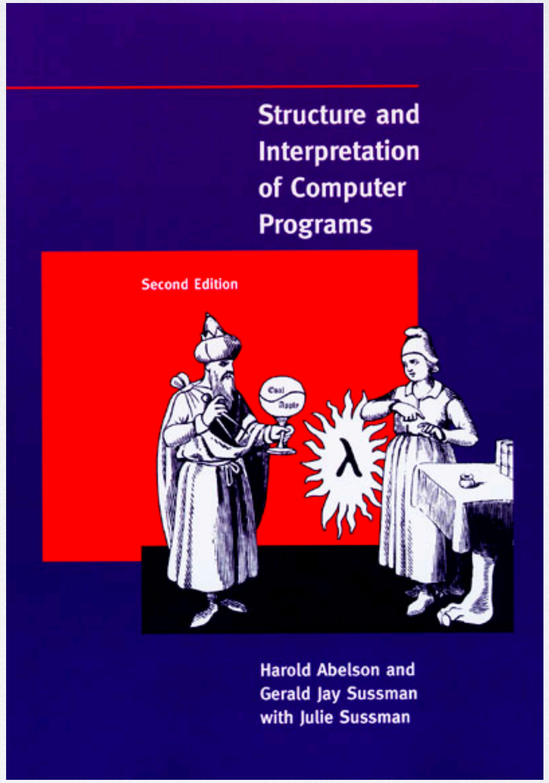
王迪

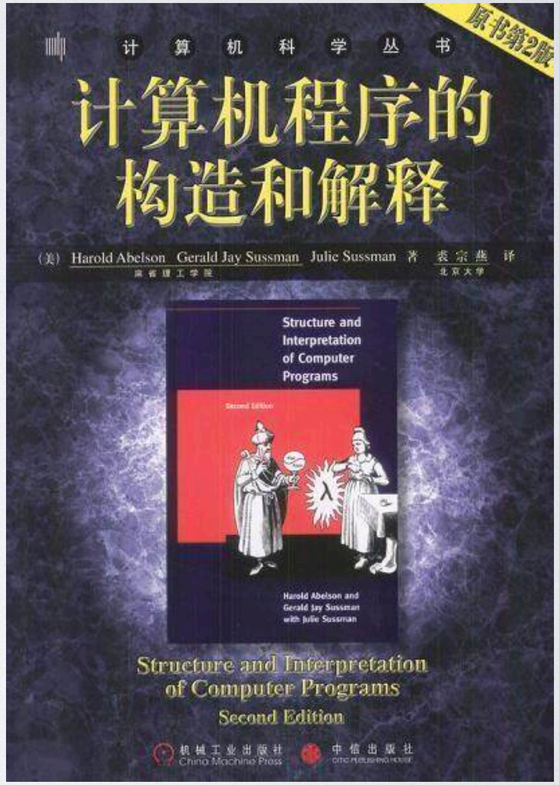
wangdi95@pku.edu.cn

2023 年 6 月 17 日

计算机科学 = 编程语言构造?

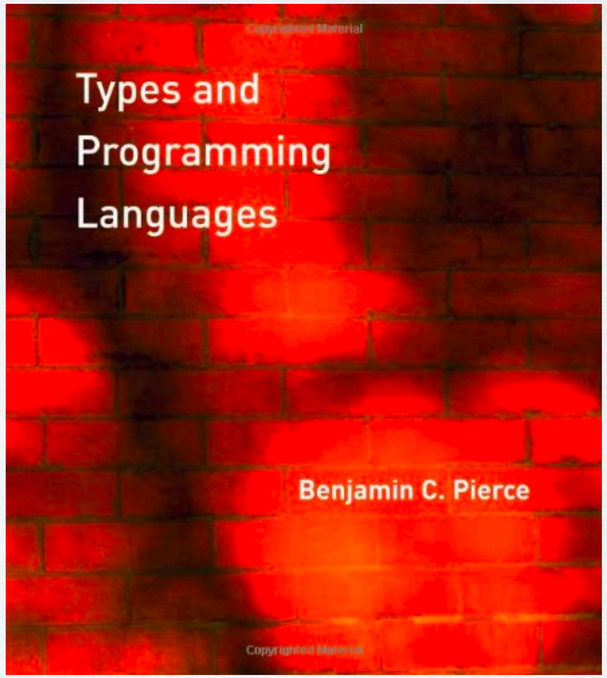
「事实上，我们几乎可以把任何程序看做是某个语言的求值器……处理大规模计算机系统的技术，与构造新的编程语言的技术有紧密的联系，而**计算机科学本身**不过（也不更少）就是关于如何**构造适当的描述语言**的学科。」

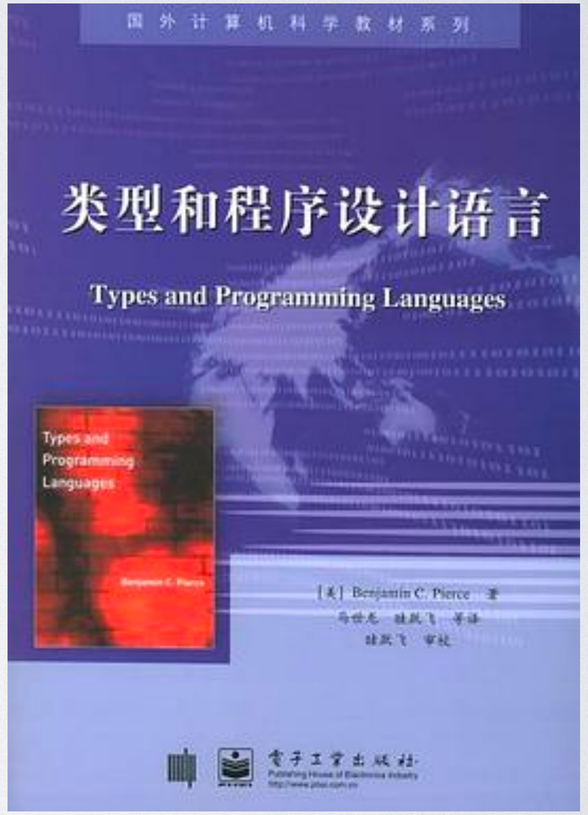




类型系统是什么？

「类型系统是指一种**根据程序所计算的值进行分类**从而**证明某些（不良）程序行为不会发生**的轻量级形式化方法。」





B 语言：C 语言的前身

```
printn(n, b) {
   extrn putchar;
   auto a;

   a = n / b;
   if (a)
      printn(a, b);
   putchar(n % b + '0');
}
```

B 语言

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- **无类型**
- **性能差**

B 语言：C 语言的前身

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

函数参数是**无类型**的
函数也没有返回类型

B 语言

B 语言：C 语言的前身

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

auto 关键字仅表示为
变量 a 预留存储空间

B 语言中只有一种类型：
所有数据都是字（word）

B 语言：C 语言的前身

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

调用的外部函数**没有函数签名**

B 语言

B 语言：C 语言的前身

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

调用的外部函数**没有函数签名**

编译时**无法检查**函数调用
是否传入了正确的参数

B 语言

B 语言：C 语言的前身

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

调用的外部函数**没有函数签名**

编译时**无法检查**函数调用
是否传入了正确的参数

运行时检查拉低了程序性能

B 语言

C 语言 = B 语言 + **类型**！

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

```
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

C 语言 = B 语言 + **类型**！

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

```
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

C 语言 = B 语言 + **类型**!

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

```
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

C 语言 = B 语言 + **类型**!

```
printn(n, b) {
  extrn putchar;
  auto a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

B 语言

```
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

C 语言 = B 语言 + **类型**！

```c
extern int putchar(int);
void printn(int n, int b) {
  int a;

  a = n / b;
  if (a)
    printn(a, b);
  putchar(n % b + '0');
}
```

C 语言

- 1972 年
- 贝尔实验室
- 通用编程语言，常被用来开发操作系统、设备驱动、协议栈等
- **静态类型**
- **性能优越**

类型系统的优点

类型系统的优点

- **检查错误**
  - 函数调用不符合签名，对浮点数进行按位与操作，……

类型系统的优点

- **检查错误**
  - 函数调用不符合签名，对浮点数进行按位与操作，……
- **提供文档**
  - 函数的类型标注提供了基本的函数文档

类型系统的优点

- **检查错误**
  - 函数调用不符合签名，对浮点数进行按位与操作，……
- **提供文档**
  - 函数的类型标注提供了基本的函数文档
- **编译优化**
  - 去除不必要的运行时检查，优化数据结构的内存布局，……

类型系统的优点

- **检查错误**
  - 函数调用不符合签名，对浮点数进行按位与操作，……
- **提供文档**
  - 函数的类型标注提供了基本的函数文档
- **编译优化**
  - 去除不必要的运行时检查，优化数据结构的内存布局，……

- **语言安全**
- **抽象机制**

在后面会介绍

类型系统与语言设计

语言设计应当同类型系统设计并行进行且互相协同

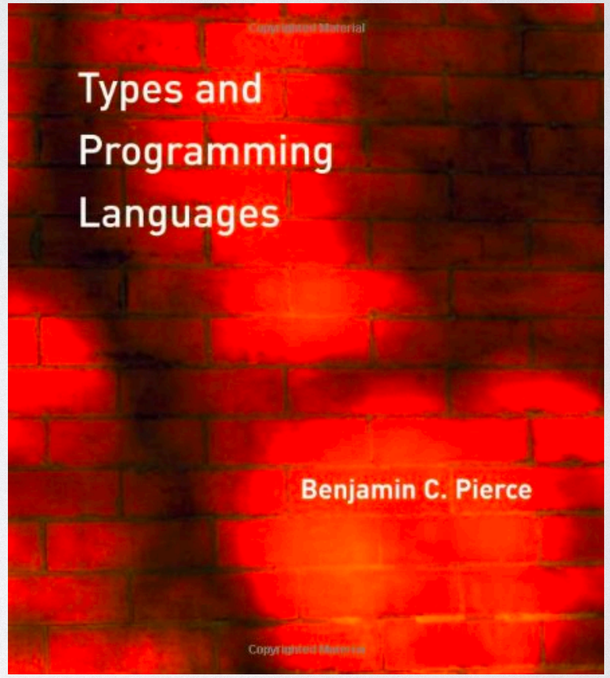
类型系统与语言设计

语言设计应当同类型系统设计并行进行且互相协同

- 不含类型系统的语言在设计时容易引入**难以进行类型检查**的编程特性
- 带类型语言的**语法可能更复杂**，需要在设计中考虑类型标注的清晰性

课程：编程语言的设计原理

- 从 2014 年开始在北大开设
- 研究生课程（早期为本研合上）
- 围绕**类型系统**这个核心组织和讲授编程语言的理论
- 教材：Types and Programming Languages

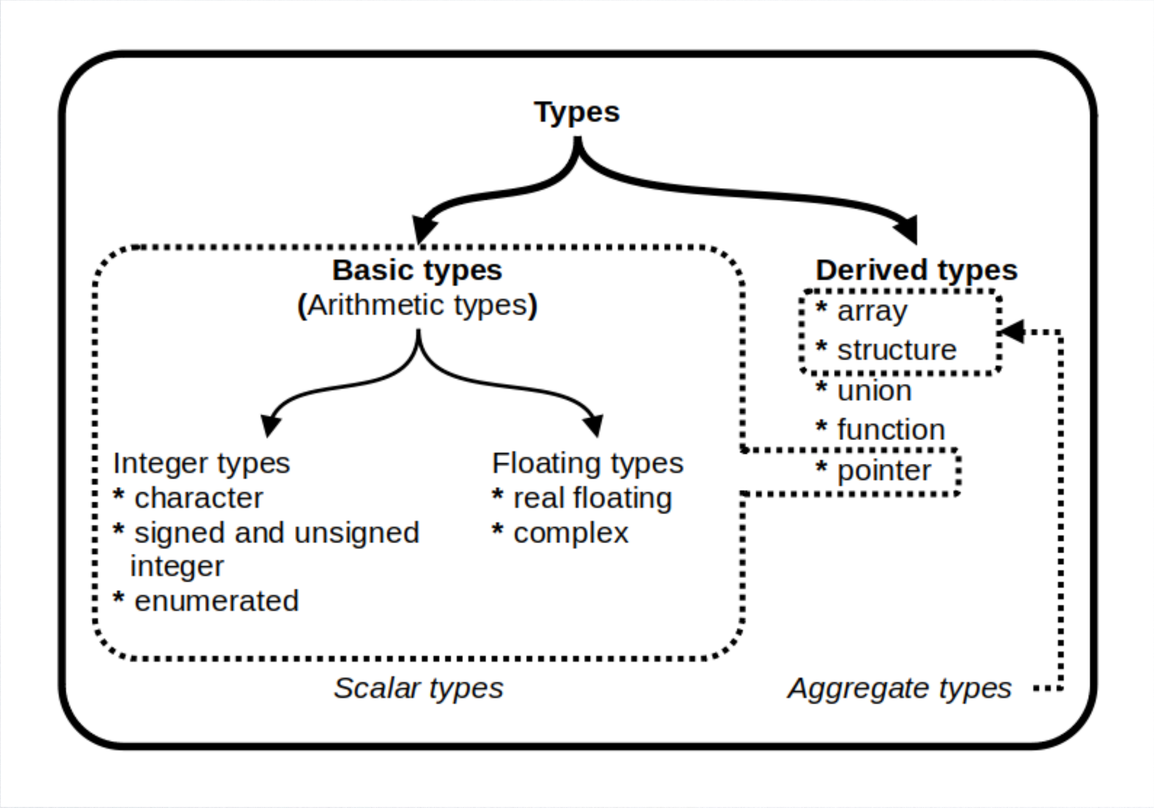


https://pku-dppl.github.io

类型主导的编程语言设计

- 编译时检查简单的程序错误
- 组织特定的编程范式或风格
- 形式化验证复杂的程序性质

C 语言



- 算术类型：整型、浮点
- 复合类型：枚举（`enum`）、结构（`struct`）、联合（`union`）
- 内存管理：数组、指针
- **弱类型**

图片来源：By Максим Пе - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=125371324

C语言

```c
extern void f(int);

struct pair {
  int a;
  int b;
};

int main() {
  float a;
  int b = a & 1;
  f();
  b = c;
  struct pair p;
  b = p.h;
  return 0;
}
```

```
t.c:10:12: error: invalid operands to binary expression ('float' and 'int')
        int b = a & 1;
                ~ ^ ~
t.c:11:4: error: too few arguments to function call, expected 1, have 0
        f();
        ~ ^
t.c:12:6: error: use of undeclared identifier 'c'
        b = c;
            ^
t.c:14:8: error: no member named 'h' in 'struct pair'
        b = p.h;
            ~ ^
4 errors generated.
```

C语言

```c
extern void f(int);

struct pair {
  int a;
  int b;
};

int main() {
  float a;
  int b = a & 1;
  f();
  b = c;
  struct pair p;
  b = p.h;
  return 0;
}
```

```
t.c:10:12: error: invalid operands to binary expression ('float' and 'int')
        int b = a & 1;
                ~ ^ ~
t.c:11:4: error: too few arguments to function call, expected 1, have 0
        f();
        ~ ^
t.c:12:6: error: use of undeclared identifier 'c'
        b = c;
            ^
t.c:14:8: error: no member named 'h' in 'struct pair'
        b = p.h;
            ~ ^
4 errors generated.
```

C语言

```c
extern void f(int);

struct pair {
  int a;
  int b;
};

int main() {
  float a;
  int b = a & 1;
  f();
  b = c;
  struct pair p;
  b = p.h;
  return 0;
}
```

```
t.c:10:12: error: invalid operands to binary expression ('float' and 'int')
        int b = a & 1;
                ~ ^ ~
t.c:11:4: error: too few arguments to function call, expected 1, have 0
        f();
        ~ ^
t.c:12:6: error: use of undeclared identifier 'c'
        b = c;
            ^
t.c:14:8: error: no member named 'h' in 'struct pair'
        b = p.h;
            ~ ^
4 errors generated.
```

C语言

```c
extern void f(int);

struct pair {
  int a;
  int b;
};

int main() {
  float a;
  int b = a & 1;
  f();
  b = c;
  struct pair p;
  b = p.h;
  return 0;
}
```

```
t.c:10:12: error: invalid operands to binary expression ('float' and 'int')
        int b = a & 1;
                ~ ^ ~
t.c:11:4: error: too few arguments to function call, expected 1, have 0
        f();
        ~ ^
t.c:12:6: error: use of undeclared identifier 'c'
        b = c;
            ^
t.c:14:8: error: no member named 'h' in 'struct pair'
        b = p.h;
            ~ ^
4 errors generated.
```

C语言

```c
extern void f(int);

struct pair {
  int a;
  int b;
};

int main() {
  float a;
  int b = a & 1;
  f();
  b = c;
  struct pair p;
  b = p.h;
  return 0;
}
```

```
t.c:10:12: error: invalid operands to binary expression ('float' and 'int')
        int b = a & 1;
                ~ ^ ~
t.c:11:4: error: too few arguments to function call, expected 1, have 0
        f();
        ~ ^
t.c:12:6: error: use of undeclared identifier 'c'
        b = c;
            ^
t.c:14:8: error: no member named 'h' in 'struct pair'
        b = p.h;
            ~ ^
4 errors generated.
```

Rust 语言

- 2015 年
- Mozilla
- 通用编程语言，强调性能、安全以及并发
- **基于静态类型的内存安全**
- **性能不错**

Rust 语言

- 2015 年
- Mozilla
- 通用编程语言，强调性能、安全以及并发
- **基于静态类型的内存安全**
- **性能不错**

类型系统的优点：

- 检查错误
- 提供文档
- 编译优化
- **语言安全**
- 抽象机制

Rust 语言

Rust 语言

```rust
// i32为32位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

Rust 语言

```rust
// i32为32位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

```rust
let x = 1; // 不可变变量

let mut y = 1; // 可变变量
y = y + 1; // 可变变量赋值
```

Rust 语言

```rust
// i32为32位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

```rust
let x = 1; // 不可变变量

let mut y = 1; // 可变变量
y = y + 1; // 可变变量赋值
```

```rust
// C风格的枚举类型
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
let up = Direction::Up;
```

Rust 语言

```rust
// i32为32位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

```rust
let x = 1; // 不可变变量

let mut y = 1; // 可变变量
y = y + 1; // 可变变量赋值
```

```rust
// C风格的枚举类型
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
let up = Direction::Up;
```

```rust
// 枚举类型可携带额外信息
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}
let two = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;
```

Rust 语言

```rust
// i32为32位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

```rust
let x = 1; // 不可变变量

let mut y = 1; // 可变变量
y = y + 1; // 可变变量赋值
```

```rust
// C风格的枚举类型
enum Direction {
    Left,
    Right,
    Up,
    Down,
}
let up = Direction::Up;
```

```rust
// 枚举类型可携带额外信息
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}
let two = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;
```

```rust
// 枚举类型的分类讨论
match two {
    OptionalI32::AnI32(n) => println!("i32: {}", n),
    OptionalI32::Nothing => println!("nothing"),
}
```

Rust 语言

Rust 语言

```
// 拥有数据所有权的指针
// 当所有者作用域结束时，自动释放内存
let mut mine: Box<i32> = Box::new(3);
*mine = 5;
```

Rust 语言

```rust
// 拥有数据所有权的指针
// 当所有者作用域结束时，自动释放内存
let mut mine: Box<i32> = Box::new(3);
*mine = 5;
```

```rust
// 转移数据所有权
let mut now_its_mine = mine;
*now_its_mine += 2;
// println!("{}", mine); // 类型检查报错
```

Rust 语言

```rust
// 拥有数据所有权的指针
// 当所有者作用域结束时，自动释放内存
let mut mine: Box<i32> = Box::new(3);
*mine = 5;
```

```rust
// 转移数据所有权
let mut now_its_mine = mine;
*now_its_mine += 2;
// println!("{}", mine); // 类型检查报错
```

```rust
// 使用Box定义链表类型
enum List {
    Nil,
    // Cons(i32, List)会报错
    // 因为无法静态计算类型大小
    Cons(i32, Box<List>),
}

use List::{Cons, Nil};
```

Rust 语言

```rust
// 拥有数据所有权的指针
// 当所有者作用域结束时，自动释放内存
let mut mine: Box<i32> = Box::new(3);
*mine = 5;
```

```rust
// 转移数据所有权
let mut now_its_mine = mine;
*now_its_mine += 2;
// println!("{}", mine); // 类型检查报错
```

```rust
// 使用Box定义链表类型
enum List {
    Nil,
    // Cons(i32, List)会报错
    // 因为无法静态计算类型大小
    Cons(i32, Box<List>),
}

use List::{Cons, Nil};
```

```rust
// l1拥有这个链表的所有权
let l1 = Cons(1, Box::new(Cons(2, Box::new(Nil))));
// 链表的所有权从l1转移到l2
let l2 = l1;
// let l3 = l1; // 类型检查报错
```

Rust 语言

Rust 语言

- 编译时的类型检查能发现多种内存安全问题
- 在不使用 `unsafe` 特性时，是**强类型**语言

Rust 语言

- 编译时的类型检查能发现多种内存安全问题
- 在不使用 `unsafe` 特性时，是**强类型**语言
- Rust 类型系统设计的理论基础：
  - 线性类型（Linear Types）
  - 区域类型（Region Types）
  - 所有权类型（Ownership Types）

TypeScript 语言

- 2012 年
- Microsoft
- 为 JavaScript 提供静态类型检查和类型标注
- **静态、动态类型结合**
- **类型推导**

```
> typeof NaN
"number"
> 9999999999999999
10000000000000000
> 0.5+0.1==0.6
true
> 0.1+0.2==0.3
false
> Math.max()
-Infinity
> Math.min()
Infinity
> []+[]
""
> []+{}
"[object Object]"
> {}+[]
0
> true+true+true===3
true
> true-true
0
> true==1
true
> true===1
false
> (!+[]+[]+![]).length
9
> 9+"1"
"91"
> 91-"1"
90
> []==0
true
```

图片来源：https://devrant.com/rants/1863017/another-javascript-joke-because-why-not-still-curious-though-which-frameworks-fo

TypeScript 语言

```
var a = [] + []

var b = 91 - "1"

var c = ([] == 0)
```

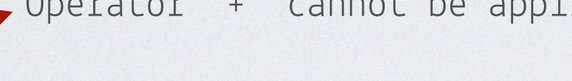
Operator '+' cannot be applied to types 'never[]' and ‘never[]'.

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

This comparison appears to be unintentional because the types 'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares objects by reference, not value.

TypeScript 语言

```
var a = [] + []

var b = 91 - "1"

var c = ([] == 0)
```



```
Operator '+' cannot be applied to types 'never[]' and ‘never[]'.

The right-hand side of an arithmetic operation must be of type 'any',
'number', 'bigint' or an enum type.

This comparison appears to be unintentional because the types
'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares
objects by reference, not value.
```

TypeScript 语言

```
var a = [] + []

var b = 91 - "1"

var c = ([] == 0)
```

Operator '+' cannot be applied to types 'never[]' and ‘never[]'.

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

This comparison appears to be unintentional because the types 'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares objects by reference, not value.

TypeScript 语言

```
var a = [] + []

var b = 91 - "1"

var c = ([] == 0)
```

Operator '+' cannot be applied to types 'never[]' and ‘never[]'.

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

This comparison appears to be unintentional because the types 'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares objects by reference, not value.

TypeScript 语言

```
var a = [] + []
```

Operator '+' cannot be applied to types 'never[]' and ‘never[]'.

```
var b = 91 - "1"
```

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

```
var c = ([] == 0)
```

This comparison appears to be unintentional because the types 'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares objects by reference, not value.

```
function successor(n: number | bigint): number | bigint {
  return ++n
}
```

并集类型（Union Types）

TypeScript 语言

```
var a = [] + []

var b = 91 - "1"

var c = ([] == 0)
```

Operator '+' cannot be applied to types 'never[]' and 'never[]'.

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.

This comparison appears to be unintentional because the types 'never[]' and 'number' have no overlap.
This condition will always return 'false' since JavaScript compares objects by reference, not value.

```
function successor(n: number | bigint): number | bigint {
  return ++n
}
```

并集类型（Union Types）

```
type Bear = { name: string } & { honey: boolean }
function greet(b: Bear) {
  return b.name + (b.honey ? " with honey" : "")
}
```

交集类型（Intersection Types）

类型主导的编程语言设计

- [x] 编译时检查简单的程序错误
- [ ] 组织特定的编程范式或风格
- [ ] 形式化验证复杂的程序性质

类型主导的编程语言设计

- ☑ 编译时检查简单的程序错误
- ☐ 组织特定的编程范式或风格
- ☐ 形式化验证复杂的程序性质

类型系统的优点：

- 检查错误
- 提供文档
- 编译优化
- 语言安全
- **抽象机制**

泛型编程

```
int apply_twice_int(function<int(int)> f, int a) {
  return f(f(a));
}

string apply_twice_string(function<string(string)> f, string a) {
  return f(f(a));
}
```

泛型编程

```
int apply_twice_int(function<int(int)> f, int a) {
  return f(f(a));
}

string apply_twice_string(function<string(string)> f, string a) {
  return f(f(a));
}
```

C++ 语言中提供的函数包装器 `Ret(Args…)`

泛型编程

```
int apply_twice_int(function<int(int)> f, int a) {
  return f(f(a));
}
string apply_twice_string(function<string(string)> f, string a) {
  return f(f(a));
}
```

```
template<typename T>
T apply_twice(function<T(T)> f, T a) {
  return f(f(a));
}
```

泛型编程

```
int apply_twice_int(function<int(int)> f, int a) {
  return f(f(a));
}
string apply_twice_string(function<string(string)> f, string a) {
  return f(f(a));
}
```

```
template<typename T>
T apply_twice(function<T(T)> f, T a) {
  return f(f(a));
}
```

T 是一个抽象的**类型变量**

泛型编程

```
static <T> T applyTwice(UnaryFunc<T, T> f, T a) {
  ...
}
static < T extends Comparable<T> > void sort(T[] a) {
  ...
}
```

Java 语言
支持带约束的泛型

泛型编程

```
static <T> T applyTwice(UnaryFunc<T, T> f, T a) {
   ...
}
static < T extends Comparable<T> > void sort(T[] a) {
   ...
}
```

Java 语言
支持带约束的泛型

```
ghci> applyTwice f a = f (f a)

ghci> :t applyTwice
applyTwice :: (t -> t) -> t -> t
```

Haskell 语言
支持泛型类型的自动推导

泛型编程

```
static <T> T applyTwice(UnaryFunc<T, T> f, T a) {
   ...
}
static < T extends Comparable<T> > void sort(T[] a) {
   ...
}
```

Java 语言
支持带约束的泛型

```
ghci> applyTwice f a = f (f a)

ghci> :t applyTwice
applyTwice :: (t -> t) -> t -> t
```

Haskell 语言
支持泛型类型的自动推导

- 泛型类型系统的理论基础：全称类型（Universal Types）

面向对象编程

面向对象编程

```
struct Shape {
  Shape() {}
  virtual float area() = 0;
};
```

抽象类作为**接口**，可以有多种实现

面向对象编程

```
struct Shape {
  Shape() {}
  virtual float area() = 0;
};
```

抽象类作为**接口**，可以有多种实现

```
struct Circle: public Shape {
  Circle(float r): r(r) {}
  float area() { return PI * r * r; }
private:
  float r;
};
```

实现一：圆形，其半径为私有数据，即进行了**封装**

面向对象编程

```
struct Shape {
  Shape() {}
  virtual float area() = 0;
};
```

抽象类作为**接口**，可以有多种实现

```
struct Circle: public Shape {
  Circle(float r): r(r) {}
  float area() { return PI * r * r; }
private:
  float r;
};
```

实现一：圆形，其半径为私有数据，即进行了**封装**

```
struct Rectangle: public Shape {
  Rectangle(float a, float b): a(a), b(b) {}
  float area() { return a * b; }
private:
  float a, b;
};
```

实现二：长方形

面向对象编程

```
struct Shape {
  Shape() {}
  virtual float area() = 0;
};
```

抽象类作为**接口**，可以有多种实现

```
struct Circle: public Shape {
  Circle(float r): r(r) {}
  float area() { return PI * r * r; }
private:
  float r;
};
```

实现一：圆形，其半径为私有数据，即进行了**封装**

```
struct Rectangle: public Shape {
  Rectangle(float a, float b): a(a), b(b) {}
  float area() { return a * b; }
private:
  float a, b;
};
```

实现二：长方形

```
struct Square: public Rectangle {
  Square(float s): Rectangle(s, s) {}
};
```

实现三：正方形，通过**继承**自长方形达成了代码复用

面向对象编程

面向对象编程

- **接口**：同一个接口可以有不同的实现
- **封装**：内部数据结构可以对外隐藏
- **继承**：达成不同类对象间的代码复用

面向对象编程

- **接口**：同一个接口可以有不同的实现
- **封装**：内部数据结构可以对外隐藏
- **继承**：达成不同类对象间的代码复用

- 类型系统可以在编译时**检查程序遵守了以上原则**
- 面向对象编程范式的理论基础：子类型（Subtyping）

函数式编程

- 通过定义和调用**函数**来完成计算
- 其风格偏向**声明式**而非命令式

函数式编程

- 通过定义和调用**函数**来完成计算
- 其风格偏向**声明式**而非命令式

```
ghci> double x = x + x
ghci> double 2
4
ghci> double (double 2)
8
```

Haskell 语言
支持类型推导

函数式编程

- 通过定义和调用**函数**来完成计算
- 其风格偏向**声明式**而非命令式

```
ghci> double x = x + x
ghci> double 2
4
ghci> double (double 2)
8
```

```
ghci> let sum [] = 0
ghci|     sum (x:xs) = x + sum xs
ghci|
ghci> sum [1..10]
55
```

Haskell 语言
支持类型推导

函数式编程

- 通过定义和调用**函数**来完成计算
- 其风格偏向**声明式**而非命令式

```
ghci> double x = x + x
ghci> double 2
4
ghci> double (double 2)
8
```

```
ghci> let sum [] = 0
ghci|     sum (x:xs) = x + sum xs
ghci|
ghci> sum [1..10]
55
```

Haskell 语言
支持类型推导

```
ghci> let qsort [] = []
ghci|     qsort (x:xs) = qsort ys ++ [x] ++ qsort zs
ghci|                 where
ghci|                   ys = [a | a <- xs, a <= x]
ghci|                   zs = [b | b <- xs, b > x]
```

函数式编程

- 函数式编程语言不一定需要静态类型系统，例如 JavaScript、Lisp 等
- 类型系统可以提供多种**函数式抽象**，比如模块

函数式编程

- 函数式编程语言不一定需要静态类型系统，例如 JavaScript、Lisp 等
- 类型系统可以提供多种**函数式抽象**，比如模块

```
ghci> module CounterADT (Counter, new, get, inc) where
ghci|   newtype Counter = C Int
ghci|   new = C 1
ghci|   get (C i) = i
ghci|   inc (C i) = C (i + 1)
ghci|

ghci> get (inc (inc new))
3
```

函数式编程

- 函数式编程语言不一定需要静态类型系统，例如 JavaScript、Lisp 等
- 类型系统可以提供多种**函数式抽象**，比如模块

```
ghci> module CounterADT (Counter, new, get, inc) where
ghci|   newtype Counter = C Int
ghci|   new = C 1
ghci|   get (C i) = i
ghci|   inc (C i) = C (i + 1)
ghci|

ghci> get (inc (inc new))
3
```

模块类型系统的理论基础：
存在类型（Existential Types）

类型主导的编程语言设计

- [x] 编译时检查简单的程序错误
- [x] 组织特定的编程范式或风格
- [ ] 形式化验证复杂的程序性质

类型主导的编程语言设计

- [x] 编译时检查简单的程序错误
- [x] 组织特定的编程范式或风格
- [ ] 形式化验证复杂的程序性质

类型系统能进行多复杂的自动验证？

Liquid Haskell

- 在 Haskell 的基础上，允许用**逻辑谓词**对类型进行精化（refinement）

Liquid Haskell

- 在 Haskell 的基础上，允许用**逻辑谓词**对类型进行精化（refinement）

```
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}

{-@ head :: NonEmpty a -> a @-}
head (x:_) = x
```

Liquid Haskell

- 在 Haskell 的基础上，允许用**逻辑谓词**对类型进行精化（refinement）

```
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}

{-@ head :: NonEmpty a -> a @-}
head (x:_) = x
```

```
{-@ abs :: Int -> {v: Int | 0 <= v} @-}
abs                 :: Int -> Int
abs n | 0 < n       = n
      | otherwise   = 0 - n
```

Liquid Haskell

- 在 Haskell 的基础上，允许用**逻辑谓词**对类型进行精化（refinement）

```
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}

{-@ head :: NonEmpty a -> a @-}
head (x:_) = x
```

```
{-@ abs :: Int -> {v: Int | 0 <= v} @-}
abs                 :: Int -> Int
abs n | 0 < n       = n
      | otherwise   = 0 - n
```

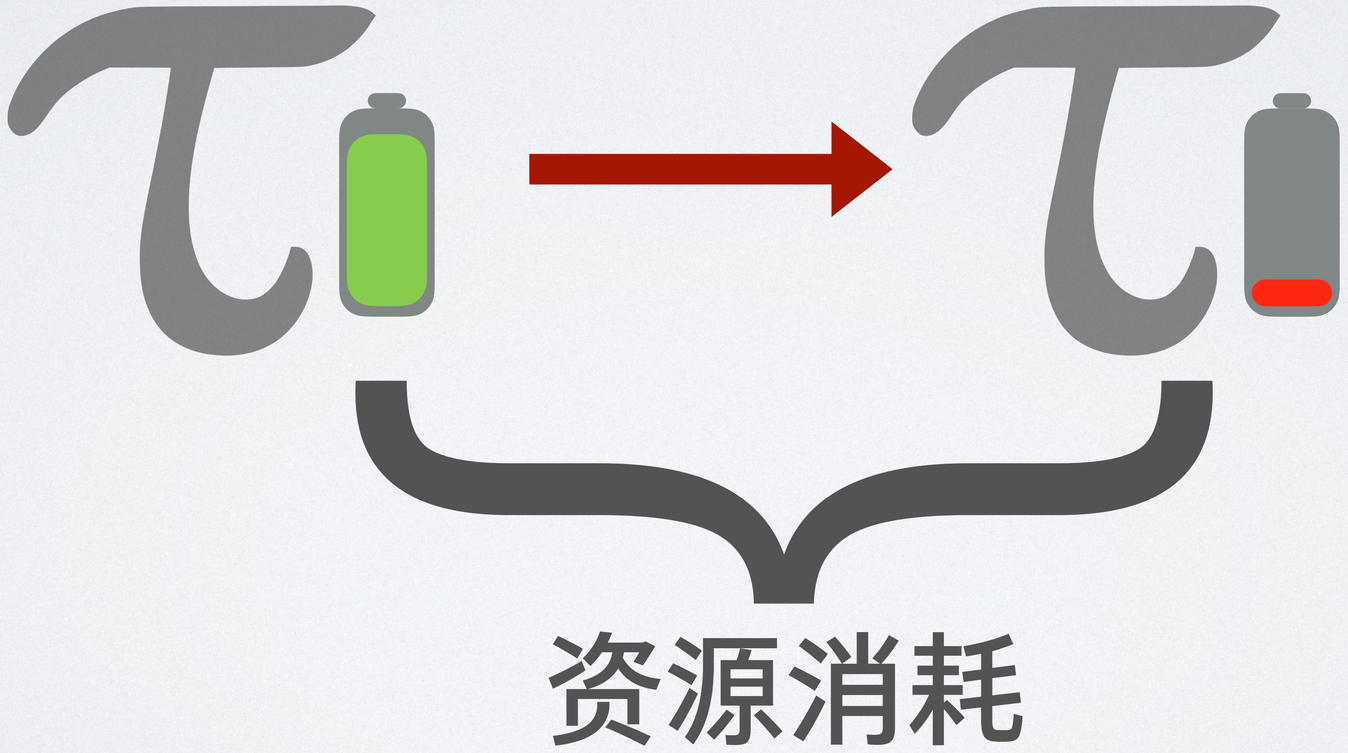
```
{-@ type OrdList a = [a]<{\xi xj -> xi <= xj}> @-}

{-@ insert :: (Ord a) => a -> OrdList a -> OrdList a @-}
insert x []     = [x]
insert x (y:ys)
  | x <= y      = x : y : ys
  | otherwise   = y : insert x ys
```

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）



Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

```
let rec append l1 l2 =
  match l1 with
  | [] ->
    l2
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

通过 tick 显式标注
程序的资源消耗模型

```
let rec append l1 l2 =
  match l1 with
  | [] ->
    l2
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

$$Cost = |\ell_1|$$

append : $\langle L^1(\alpha) \times L^0(\alpha), 0\rangle \to \langle L^0(\alpha), 0\rangle$

```
let rec append l1 l2 =
  match l1 with
  | [] ->
    l2
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

通过 tick 显式标注程序的资源消耗模型

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

$Cost = |\ell_1|$

append : $\langle L^1(\alpha) \times L^0(\alpha), 0\rangle \to \langle L^0(\alpha), 0\rangle$

```
let rec append l1 l2 =
  match l1 with
  | [] ->
    l2
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

Resource-aware ML

- 在函数式编程语言 OCaml 的基础上，通过**势能类型**分析资源消耗（如时间复杂度）

$Cost = |\ell_1|$

append : $\langle L^1(\alpha) \times L^0(\alpha), 0\rangle \to \langle L^0(\alpha), 0\rangle$

```
let rec append l1 l2 =
  match l1 with
  | [] ->
    l2
  | x::xs ->
    let () = tick(1) in
    let rest = append xs l2 in
    x::rest
```

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

类型主导的编程语言设计

- 编译时检查简单的程序错误
- 组织特定的编程范式或风格
- 形式化验证复杂的程序性质

类型主导的编程语言设计

语言设计应当同类型系统设计并行进行且互相协同

类型系统的优点：

- 检查错误
- 提供文档
- 编译优化
- 语言安全
- 抽象机制

- ☑ 编译时检查简单的程序错误
- ☑ 组织特定的编程范式或风格
- ☑ 形式化验证复杂的程序性质