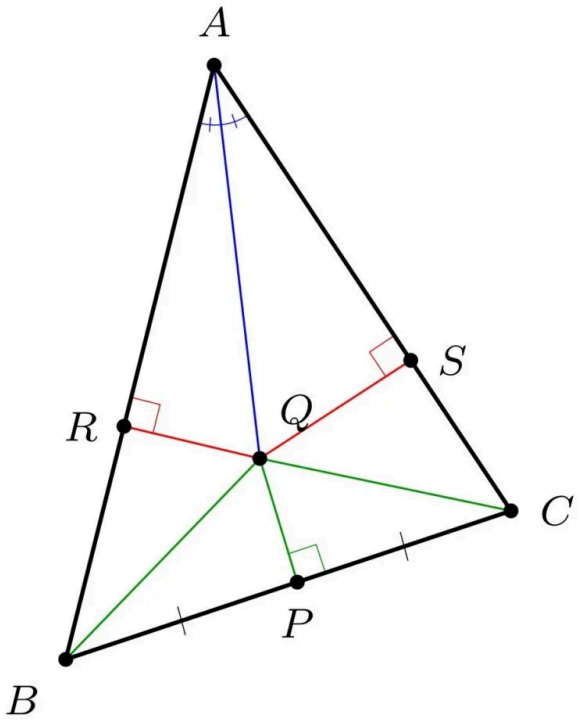
数学视角的程序

数千年来……

数学的“严格性”都是由人类保证的

- 数学期刊上的伪证、数学试卷上的一本正经胡说八道……
- 经典案例：“所有三角形都是等腰三角形”



程序的本质

程序是一种 “数学严格” 的对象

- **Everything is a state machine**
  - 程序 = 初始状态 + 迁移函数
  - 在这个视角下，程序和数学对象已经无限接近了

$$f(s) = s'$$

然而，人类并不擅长 “数学严格”

- 我们经常写出 “似是而非” 的代码
- 类似于细节有错但可以修正的数学证明
  - `for (int j = 0; j < n; i++)`
  - (有时候也会疏忽，导致设计全错)

程序的本质 (cont'd)

为什么会有程序?

- 是因为我们有无情的执行指令的机器 (计算机)
- 只有程序才配得上它

程序天生是 “人类” 的，也是 “反人类” 的

- 人类的一面：程序连接了人类世界需求
  - 我们并不是在实现 “uniform-random” 的 f
- 反人类的一面：程序会在机器上执行 😂
  - 初学者对 “机器严格” 普遍不太适应
  - 部分原因是对程序的行为没有 100% 的掌控

当我们谈论数学的时候，我们想谈论什么？

证明程序正确性！

```python
@dataclass
class AlipayAccount:
    b: int = 0  # Balance

    def deposit(...): ...
    def withdraw(...): ...
    def transfer(...): ...
```

我们可以用数学的语言提出程序的规约

- 例如：任意时刻 $a.b \geq 0$
- 有没有可能真正 “证明” 它呢?

程序正确性证明的两种方法

暴力枚举

- 写一个 driver code，运行所有可能的函数调用序列
  - PL/SE 已经研究生成 driver code 几十年了
  - assert($b \geq 0$)
- 如果机器和 driver 都没有 bug，程序就是对的

写出证明

- For all f-reachable states, $b \geq 0$ holds.
- 为 f 写一份数学证明就行了
  - 就像你在上数学课时做的习题一样

是的，我们可以“写证明”

而且还有另一个程序能帮我们检查！

- 它会拒绝一切伪证 (假设它没有 bug)
  - EMMS Lecture by Terence Tao

```
Example ceval_example1:
  empty_st =[
     X := 2;
     if (X ⩽ 1)
       then Y := 3
       else Z := 4
     end
  ]⇒ (Z !→ 4 ; X !→ 2).
Proof.
  (* We must supply the intermediate state *)
  apply E_Seq with (X !→ 2).
  - (* assignment command *)
    apply E_Asgn. reflexivity.
  - (* if command *)
    apply E_IfFalse.
    reflexivity.
    apply E_Asgn. reflexivity.
Qed.
```

```
(1/2)
beval (X !→ 2) <{ X ⩽ 1 }> = false
```

```
(2/2)
(X !→ 2) =[ Z := 4 ]⇒ (Z !→ 4; X !→ 2)
```

一些 Insights

暴力枚举带来的启发

- 我们应该把需要证明的性质写成 assertions
- `assert(u->prev->next == u)`
  - 至少可以避免 “悄悄出错” 的情况发生

写出证明带来的启发

- 容易**阅读** (self-explain) 的代码是好代码
  - 能把代码和需求联系起来
- 容易**验证** (self-evident) 的代码是好代码
  - 能把代码和正确性证明联系起来
  - “Proof-carrying code”

为操作系统建模

复习：操作系统的两个视角

应用视角 (自顶向下)

- 操作系统 = 对象 + API
  - 应用通过 syscall 访问操作系统

机器视角 (自底向上)

- 操作系统 = C 程序
  - 运行在计算机硬件上的一个普通程序

为操作系统建模

操作系统 = 状态机的管理者

- 当然，它自己也是状态机，有自己的状态

有了一个有趣的想法……

- 能不能我们自己定义 “状态机”
  - 用我们喜欢的语言、喜欢的方式
  - 不要受限于 C、汇编……
- 自己模拟状态机的执行
  - 不就有了一个 “玩具操作系统” 吗?

为操作系统建模 (cont'd)

简化的操作系统模型

- 用更方便的编程语言描述状态机
  - 依然是程序
  - 依旧是 “数学严格” 的对象
- 但用更简单的方法实现操作系统
  - 管理状态机
  - 执行系统调用

表示状态机：当然是程序

Life is short, you need Python!

```
def StateMachine():
    b = sys_read()

    if b == 0:
        sys_write('I got a zero.')
    else:
        sys_write('I got a one.')

def main():
    sys_spawn(StateMachine)
```

玩具操作系统

操作系统中的对象

- 状态机 (进程)
  - Python 代码
  - 初始时，仅有一个状态机 (main)
  - 允许执行计算或 read, write, spawn 系统调用
- 一个进程间共享的 buffer (“设备”)

系统调用

- read(): 返回随机的 0 或 1
- write(s): 向 buffer 输出字符串 `s`
- spawn(f): 创建一个可运行的状态机 `f`

如何实现？ 🌶️

难点是多状态机的管理

- 如何在状态机之前来回切换
- 实现我们单 CPU 上运行多个程序的效果？

一些途径

- SimpleC 模拟器支持 “单步” 功能
  - 创建多个模拟器对象单步执行 (J2ME KVM 就是如此)
  - 或者干脆偷懒，启动多个 pdb
- 是否有语言机制能 “暂存” 函数的运行状态，并且之后回复？
  - 有：Generators/Coroutines

于是，我们有了“操作系统”！

30 行代码讲完《操作系统》

- 进程
- 系统调用
- 上下文切换
- 调度

你会在 Linux Kernel 中看到“类似”的代码

- “procs” → cpu->runqueue
- “current” → current = (current_thread_info()->task)

玩具的意义

玩具实现了最重要的机制

- 状态机管理、系统调用、上下文切换
- 许多更复杂的机制只是 “更多的代码”

我们甚至没有脱离真实的操作系统

```
void sys_write(const char *s) { printf("%s", s); }
int sys_read() { return rand() % 2; }
void sys_spawn(void *(*fn)(void *), void *args) {
    pthread_create(&procs[n++], NULL, fn, args);
}
```

- 学习路线：先理解玩具
- 再理解真实系统和玩具的差异

并且，打开潘多拉的盒子……

因为 spawn 的存在，操作系统中有多个状态机 (进程)

```
def Process(name):
    for _ in range(5):
        sys_write(name)

def main():
    sys_spawn(Process, 'A')
    sys_spawn(Process, 'B')
```

- 操作系统会 “雨露均沾” 地运行它们
- 但 buffer 是所有状态机共享的
  - 于是有了并发……
  - **操作系统是最早的实用并发程序**

Mosaic Model and Checker

数学视角的操作系统

状态

- 多个 “应用程序” 状态机
  - 当然，可以是模型

初始状态

- 仅有一个 “main” 状态机
  - 这个状态机处于初始状态

迁移

- 选择一个状态机执行一步
  - 就像我们在操作系统模型上看到的那样

计算机系统中的不确定性 (non-determinism)

调度：状态机的选择不确定

- `current = random.choice(self.procs)`
- 操作系统每次可以随机选择一个状态机执行一步

I/O：系统外的输入不确定

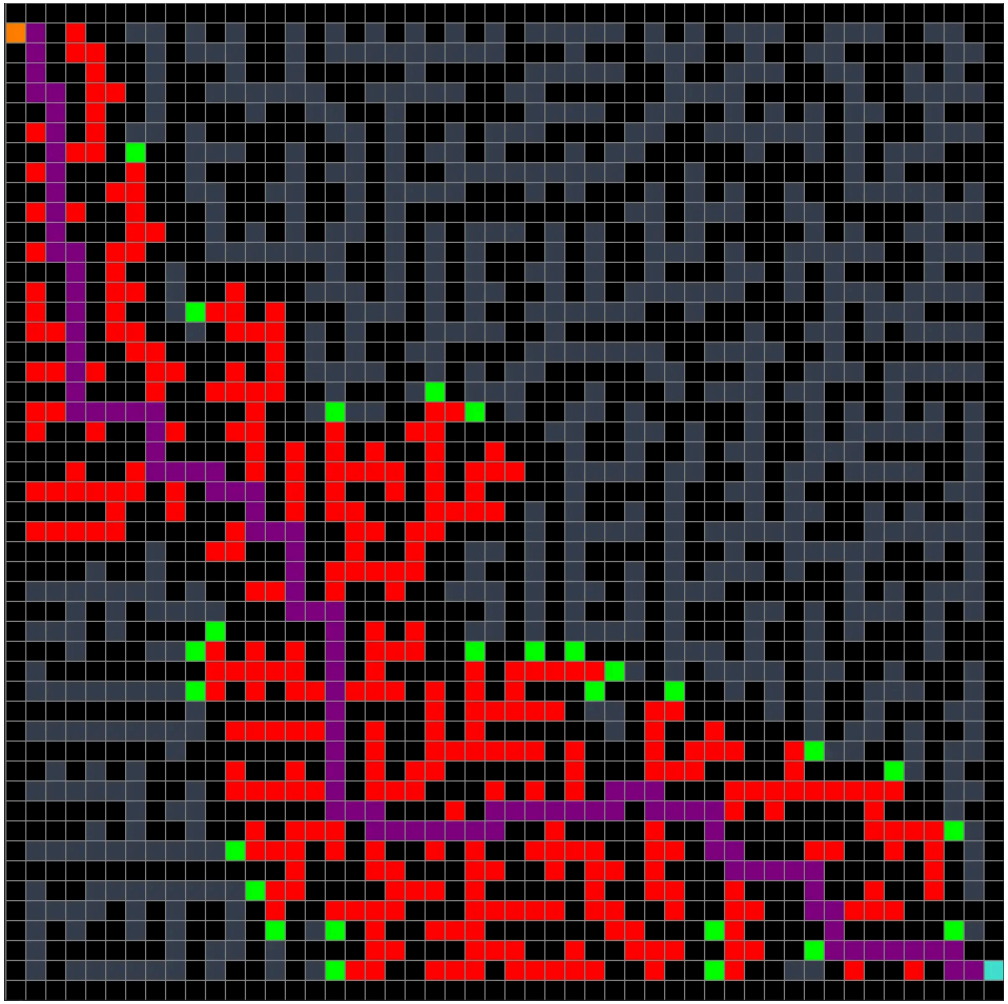
- read 返回的结果也有两种可能性
- `t = sys_read()` 后，可能 $t = 0$ 或 $t = 1$

推论：我们得到了状态图

- $u \to v \Leftrightarrow u$ 可以通过一步迁移到达 v
- 当然，我们只关系 s_0 可达的状态

你们学过的算法忽然更有用了

Breadth-first search 可以构建 “状态图”



Prove by Brute-force

想要证明程序的性质?

- 只要稍微 “修改” 一下模拟器的实现就行了

构建状态图，检查程序正确性

- read()：创建两个状态，分别是 $r = 0$ 和 $r = 1$
- 调度：为每个进程 p 创建一个状态，对应选择 p 执行
- 程序正确：不存在从 s_0 可达的 “坏状态”
  - 例如：最终 buffer 中 A 和 B 的数量相同
  - “模型检查器”；Turing Award Lecture

Putting Them Together

模型

- 理论上，我们可以建模任何系统调用
- 当然，我们选择建模**最重要**的那些
  - Three Easy Pieces!

检查器

- 最简单的 BFS 就行 (只要能获得状态机的状态)

可视化

- 我们就是绘制一个顶点是状态的图 $G(V, E)$

于是，我们有了一个更复杂的玩具

模块	系统调用	行为
基础	**choose(xs)**	返回一个 xs 中的随机的选择
基础	**write(s)**	向调试终端输出字符串 s
并发	**spawn(fn)**	创建从 fn 开始执行的线程
并发	**sched()**	切换到随机的线程/进程执行
虚拟化	fork()	创建当前状态机的完整复制
持久化	bread(k)	读取虚拟设磁盘块 k 的数据
持久化	bwrite(k, v)	向虚拟磁盘块 k 写入数据 v
持久化	sync()	将所有向虚拟磁盘的数据写入落盘
持久化	crash()	模拟系统崩溃