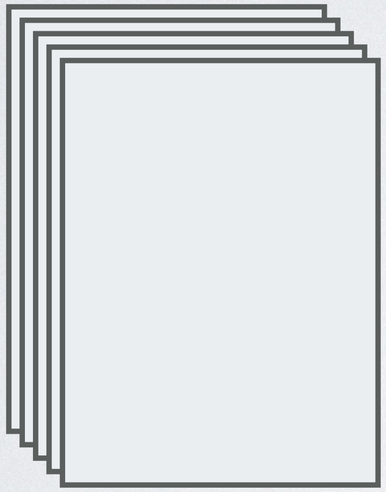
资源安全的系统编程语言

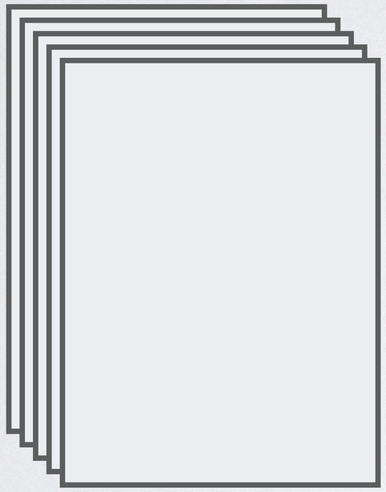
王迪
北京大学

程序的资源消耗



程序

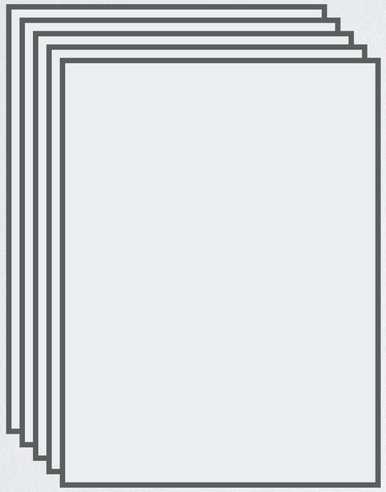
程序的资源消耗



程序

性能

程序的资源消耗

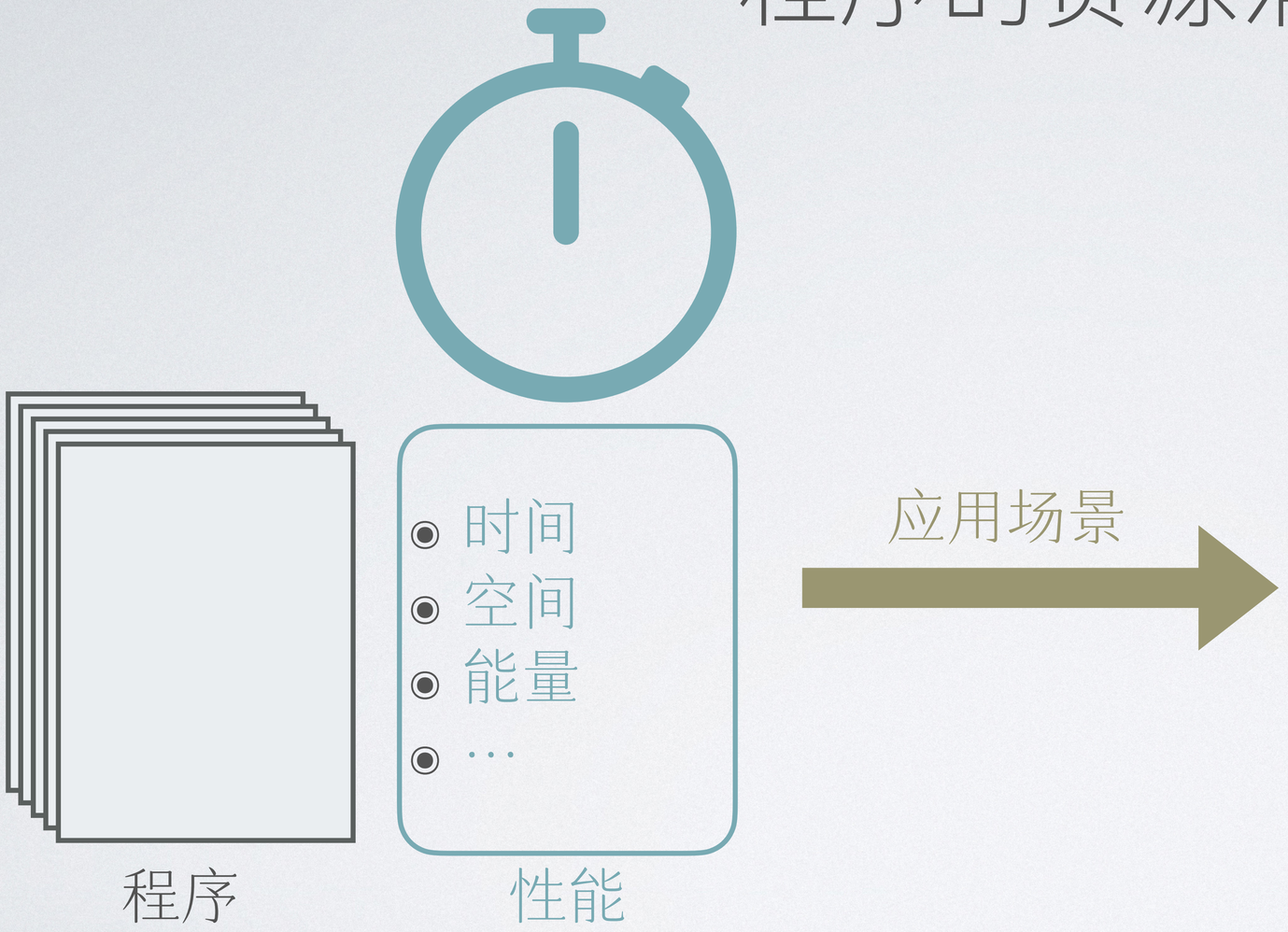


- 时间
- 空间
- 能量
- …

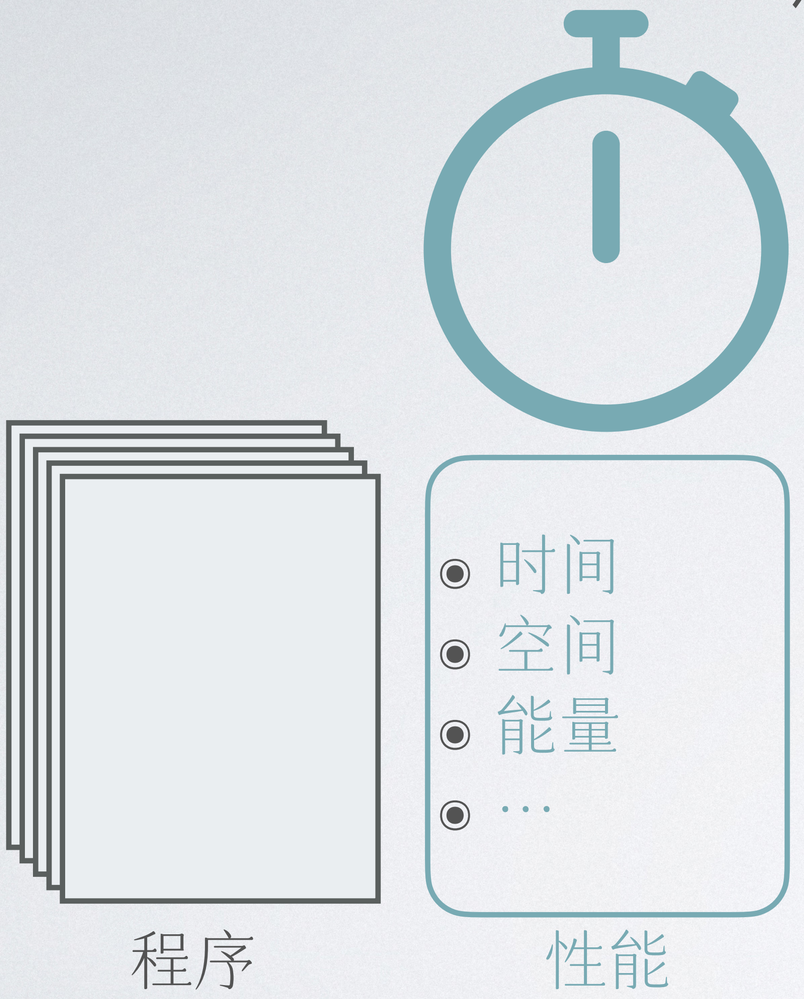
程序

性能

程序的资源消耗



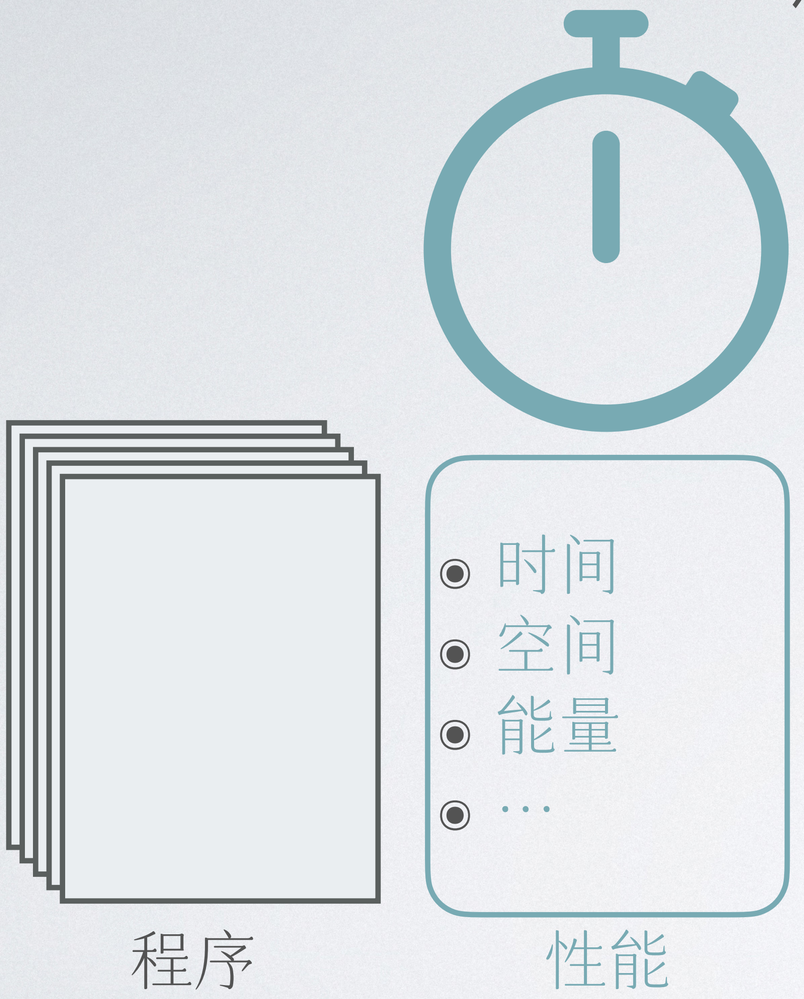
程序的资源消耗



应用场景

- 分析算法的时间复杂度

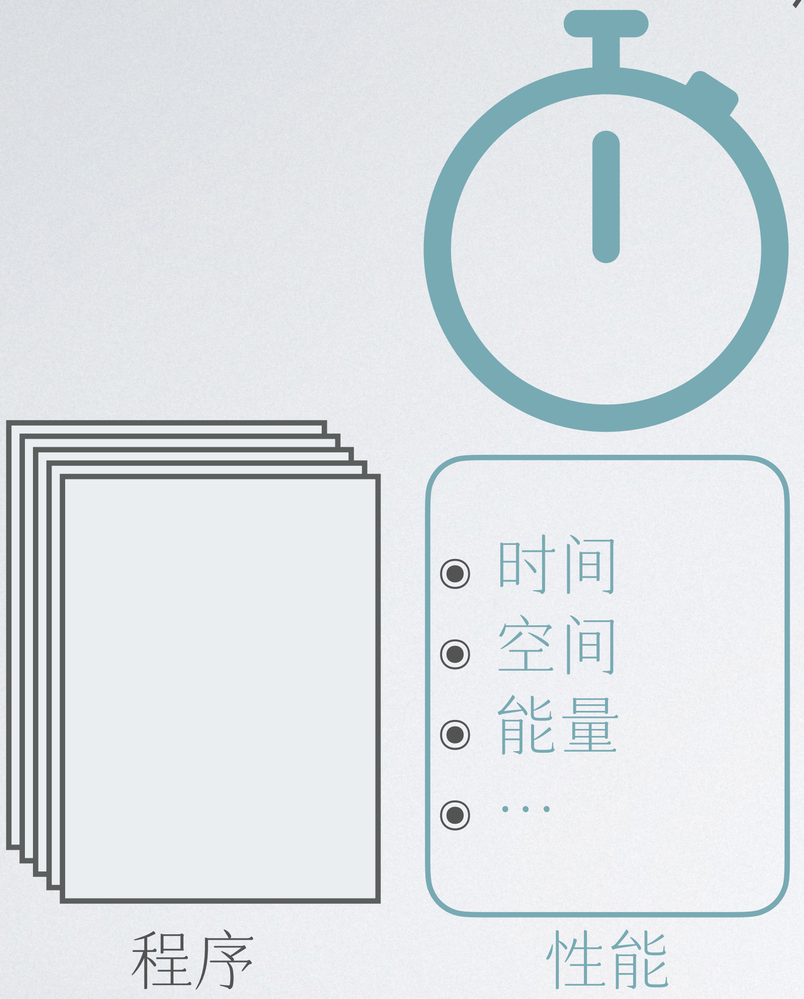
程序的资源消耗



应用场景

- 分析算法的时间复杂度
- 预测智能合约的燃气消耗

程序的资源消耗



应用场景

- 分析算法的时间复杂度
- 预测智能合约的燃气消耗
- 发现软件的拒绝服务漏洞

分析算法的时间复杂度

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

```java
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
```

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

```java
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
```

$$8|list| + 16 = O(|list|)$$

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

```
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
```

$$8|list| + 16 = O(|list|)$$

```
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
    print(list); // new function call
  }
}
```

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

```
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
```

$8|list| + 16 = O(|list|)$

```
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
    print(list); // new function call
  }
}
```

$O(|list|^2)$

该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

分析算法的时间复杂度

```java
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
```

```java
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
    print(list); // new function call
  }
}
```

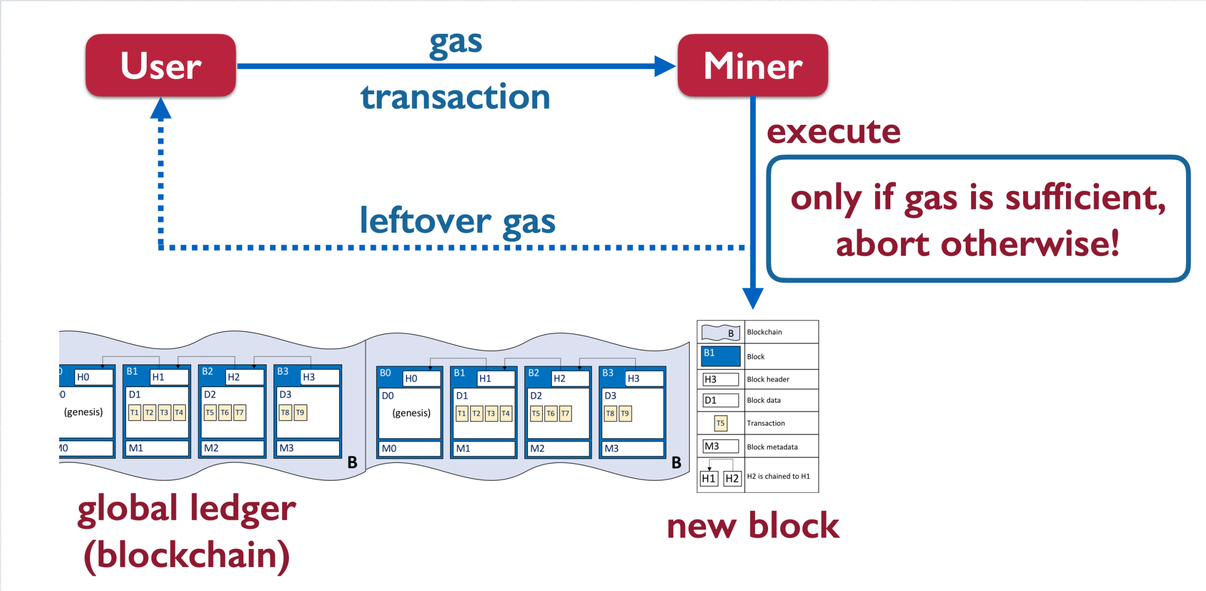
$$8|list| + 16 = O(|list|)$$

时间复杂度变高了！

$$O(|list|^2)$$

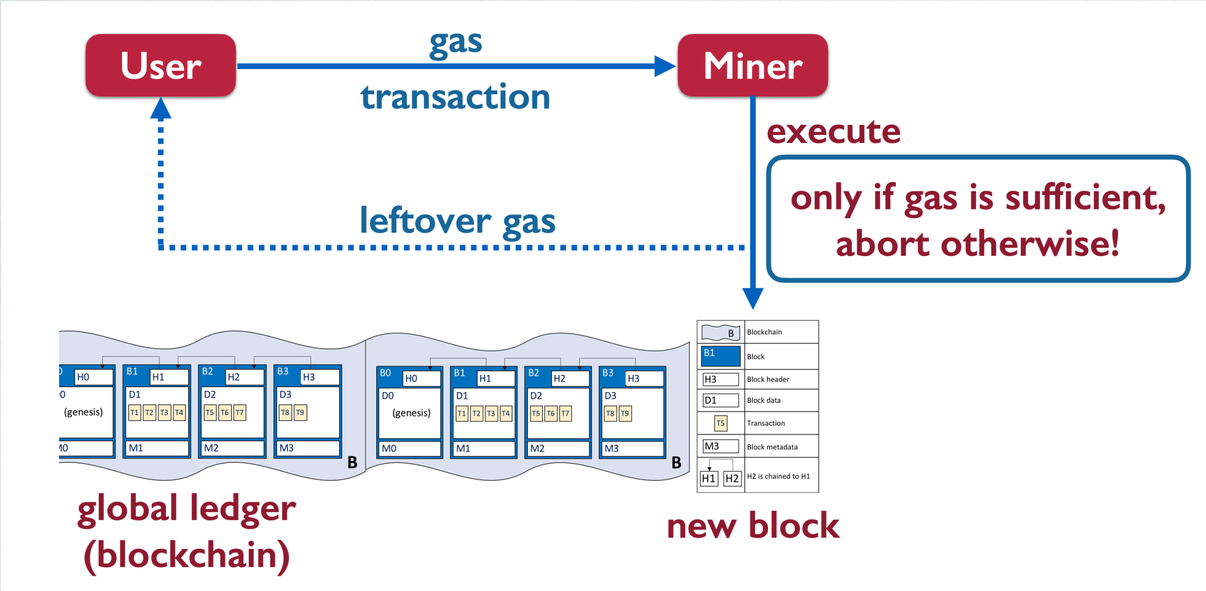
该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/

预测智能合约的燃气消耗



图片和示例来自 SAS 2020 的报告和论文：A. Das and S. Qadeer, Exact and Linear-Time Gas-Cost Analysis.

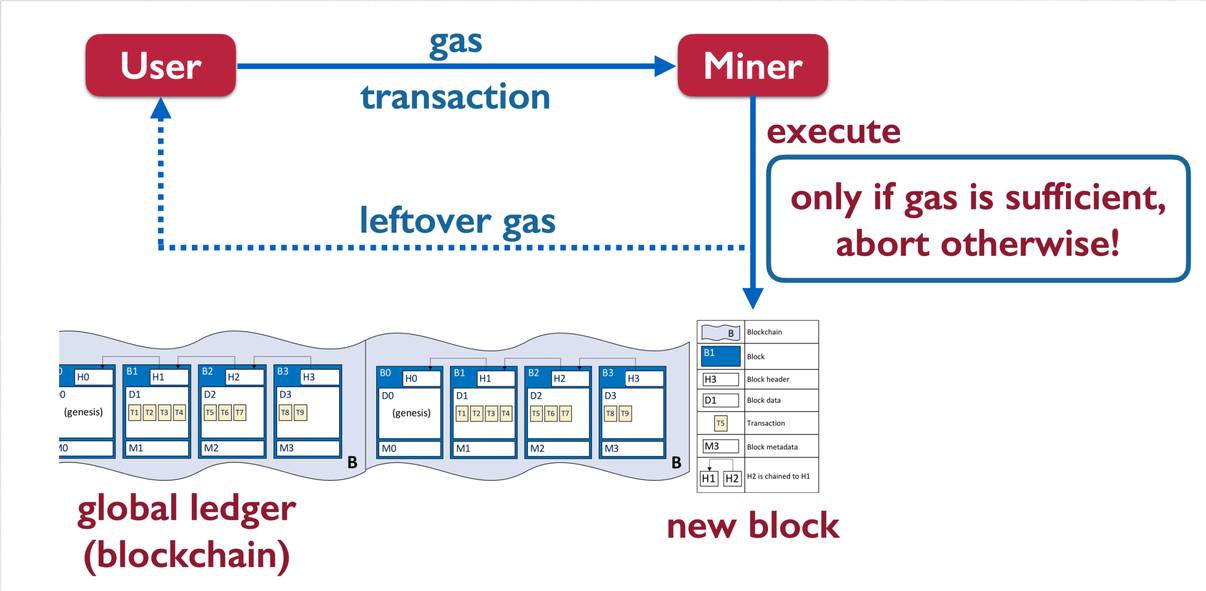
预测智能合约的燃气消耗




```
resource GasBalance {
  balance : Coin,
  gas : Gas(65)       // utilized to pay interest periodically
}
resource Bank {
  nogas_accounts : Map<address, Coin>,
  gas_accounts : Map<address, GasBalance>
}
fn [201] recharge(bank : &Bank)
fn [29] payInterest(bank : &Bank)
fn [34] signup(bank : &Bank, amount : Coin)
fn [122] balance(bank : &Bank) -> int
fn [148] deposit(bank : &Bank, amount : Coin)
fn [187] withdraw(bank : &Bank, amount : int) -> Coin
```

图片和示例来自 SAS 2020 的报告和论文：A. Das and S. Qadeer, Exact and Linear-Time Gas-Cost Analysis.

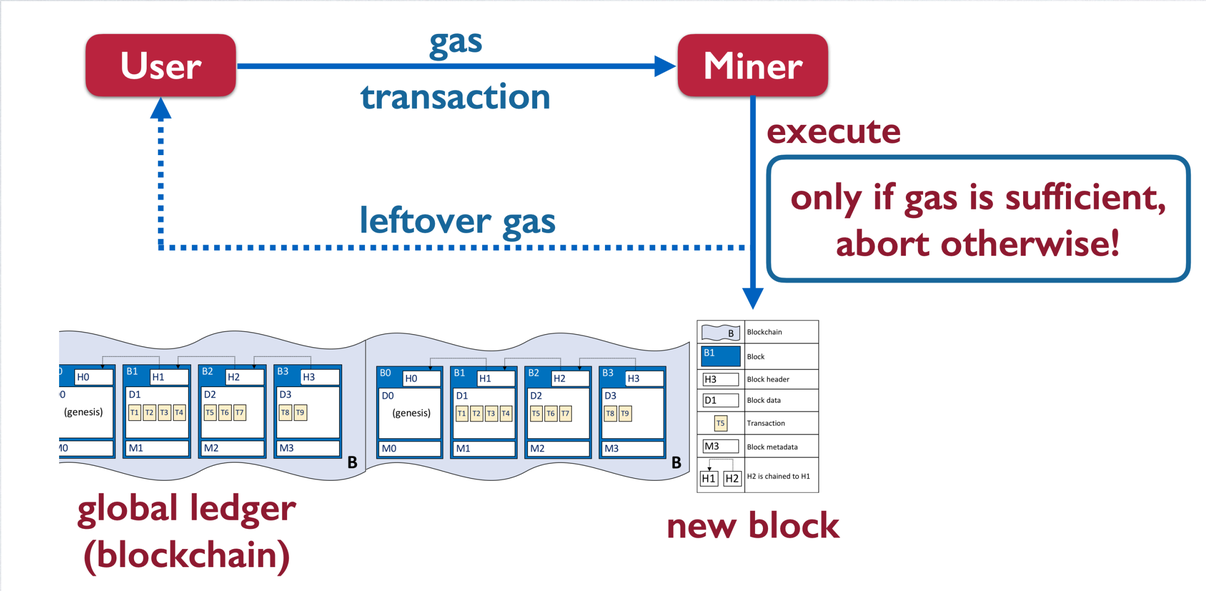
预测智能合约的燃气消耗




```
resource GasBalance {
  balance : Coin,
  gas : Gas(65)        // utilized to pay interest periodically
}
resource Bank {
  nogas_accounts : Map<address, Coin>,
  gas_accounts : Map<address, GasBalance>
}
fn [201] recharge(bank : &Bank)
fn [29]  payInterest(bank : &Bank)
fn [34]  signup(bank : &Bank, amount : Coin)
fn [122] balance(bank : &Bank) -> int
fn [148] deposit(bank : &Bank, amount : Coin)
fn [187] withdraw(bank : &Bank, amount : int) -> Coin
```

图片和示例来自 SAS 2020 的报告和论文：A. Das and S. Qadeer, Exact and Linear-Time Gas-Cost Analysis.

预测智能合约的燃气消耗



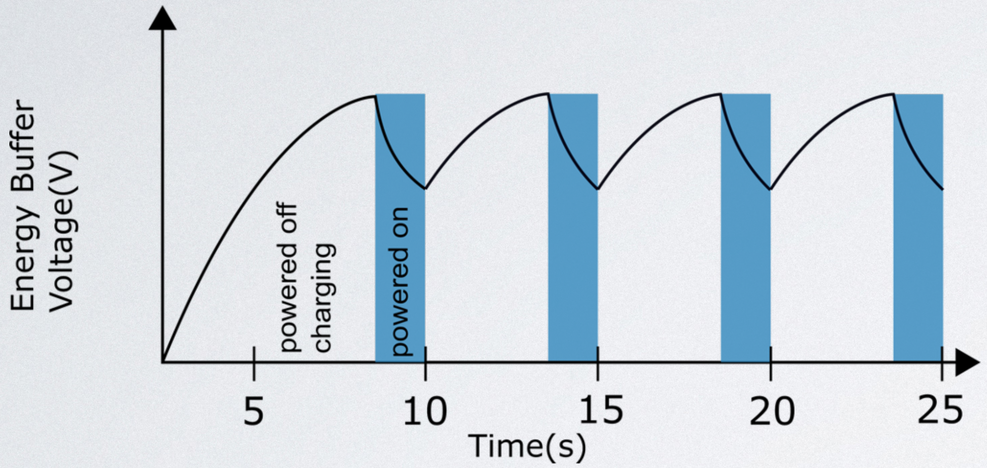

```
resource GasBalance {
  balance : Coin,
  gas : Gas(65)        // utilized to pay interest periodically
}
resource Bank {
  nogas_accounts : Map<address, Coin>,
  gas_accounts : Map<address, GasBalance>
}
fn [201] recharge(bank : &Bank)
fn [29] payInterest(bank : &Bank)
fn [34] signup(bank : &Bank, amount : Coin)
fn [122] balance(bank : &Bank) -> int
fn [148] deposit(bank : &Bank, amount : Coin)
fn [187] withdraw(bank : &Bank, amount : int) -> Coin
```

通过静态分析计算精确的燃气消耗数值

Time is Money!

图片和示例来自 SAS 2020 的报告和论文：A. Das and S. Qadeer, Exact and Linear-Time Gas-Cost Analysis.

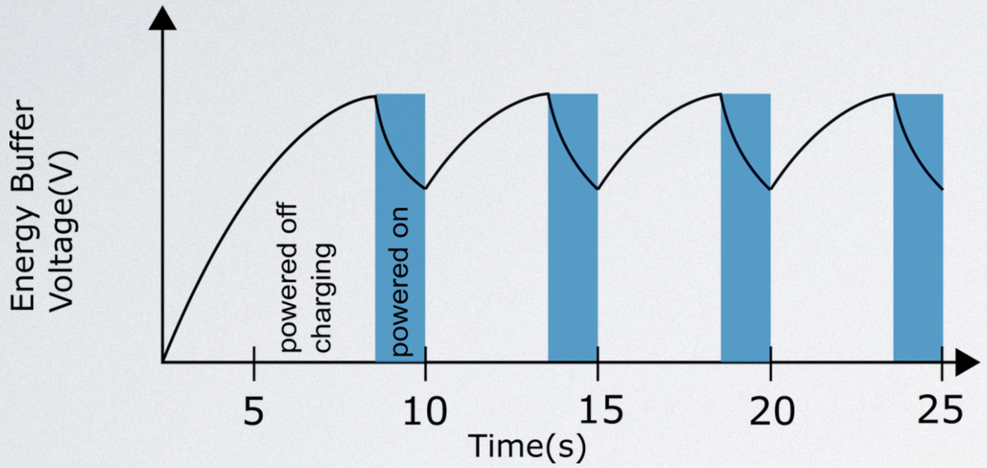
从燃气消耗到更物理的资源度量



一类物联网设备
充电与计算间歇进行

图片来自 CC 2018 的论文：A. Colin and B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs.

从燃气消耗到更物理的资源度量

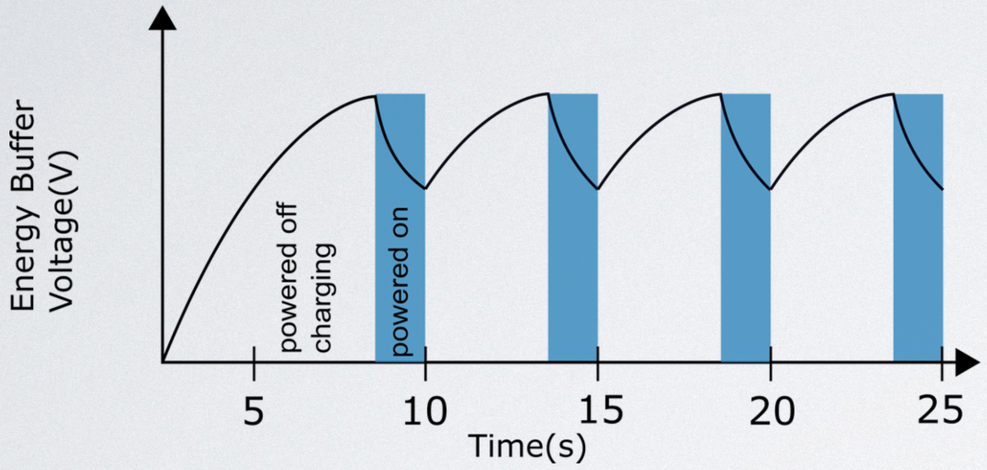


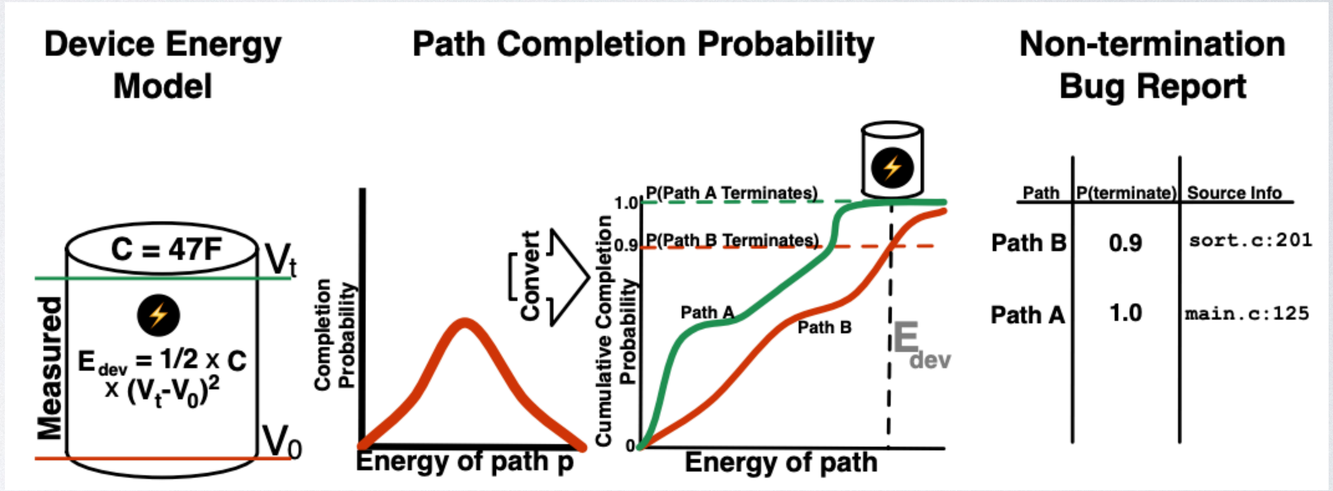
一类物联网设备
充电与计算间歇进行

每个计算任务需要在电池
容量内完成以保证一致性

图片来自 CC 2018 的论文：A. Colin and B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs.

从燃气消耗到更物理的资源度量



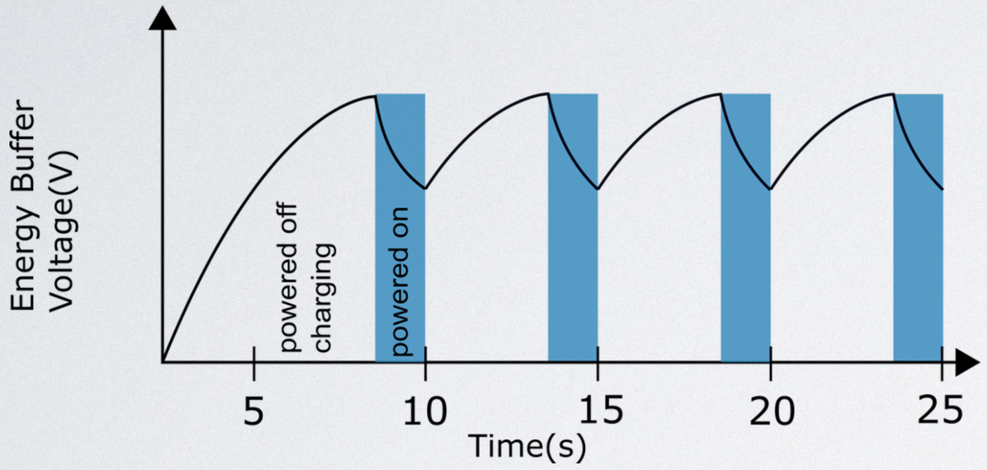


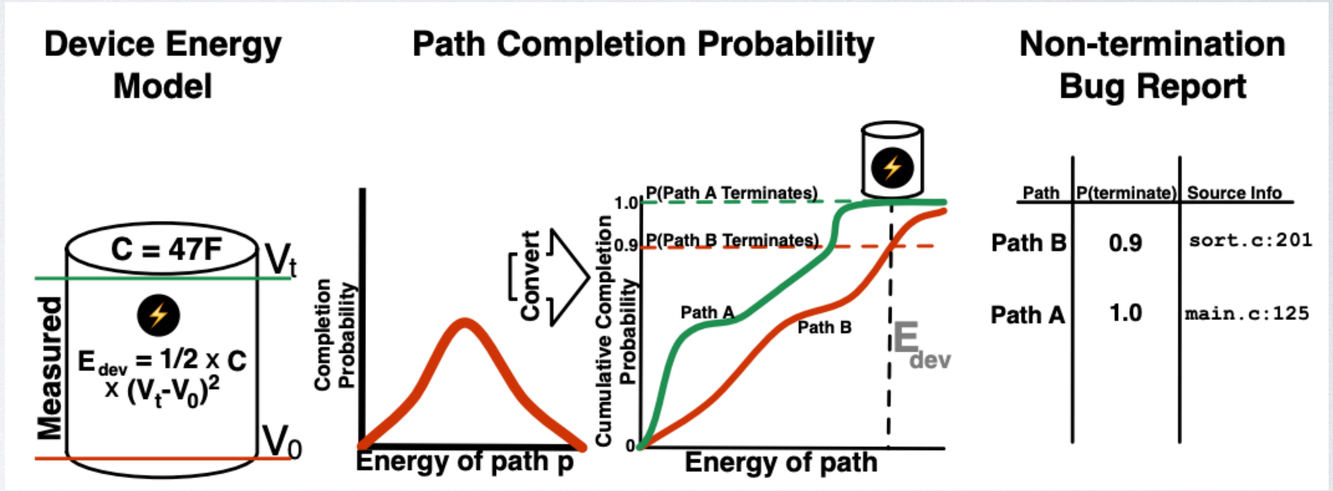
一类物联网设备
充电与计算间歇进行

每个计算任务需要在电池
容量内完成以保证一致性

图片来自 CC 2018 的论文：A. Colin and B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs.

从燃气消耗到更物理的资源度量





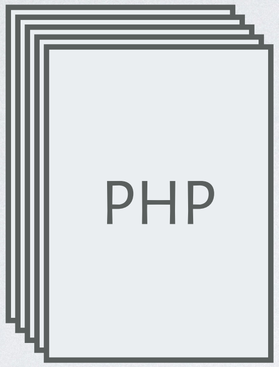
一类物联网设备
充电与计算间歇进行

每个计算任务需要在电池
容量内完成以保证一致性

对程序消耗的能量
进行概率分析

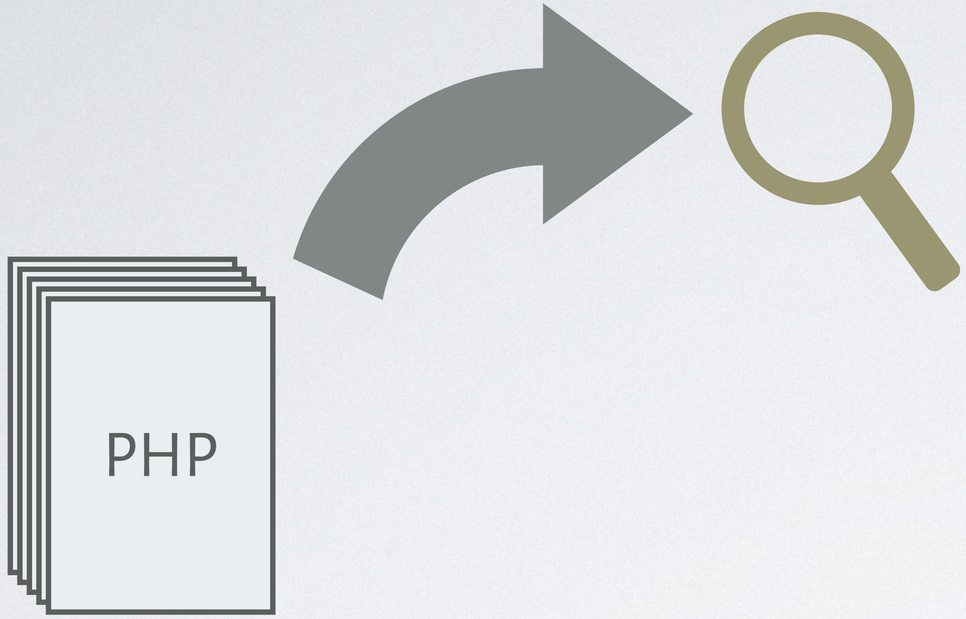
图片来自 CC 2018 的论文：A. Colin and B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs.

发现软件的拒绝服务漏洞



[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

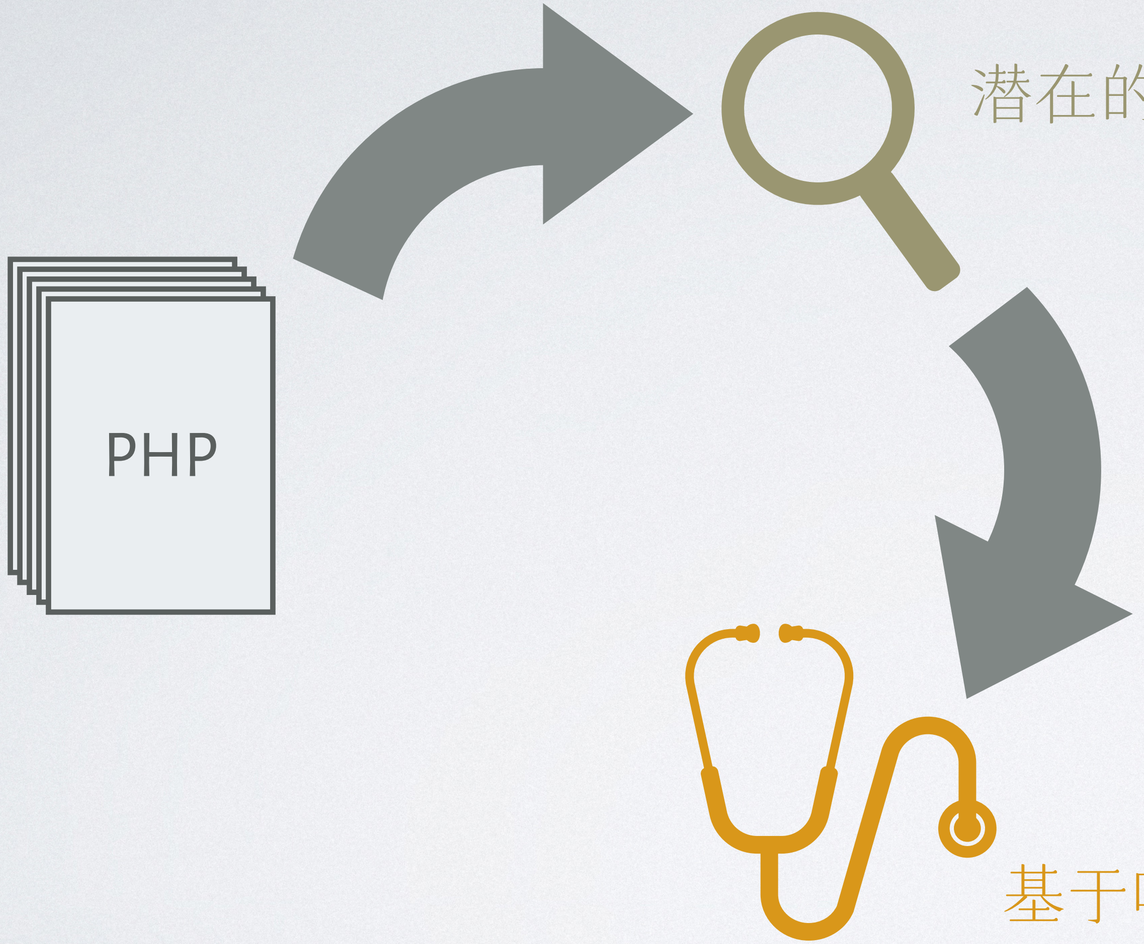
发现软件的拒绝服务漏洞



潜在的拒绝服务漏洞 [1]

[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

发现软件的拒绝服务漏洞

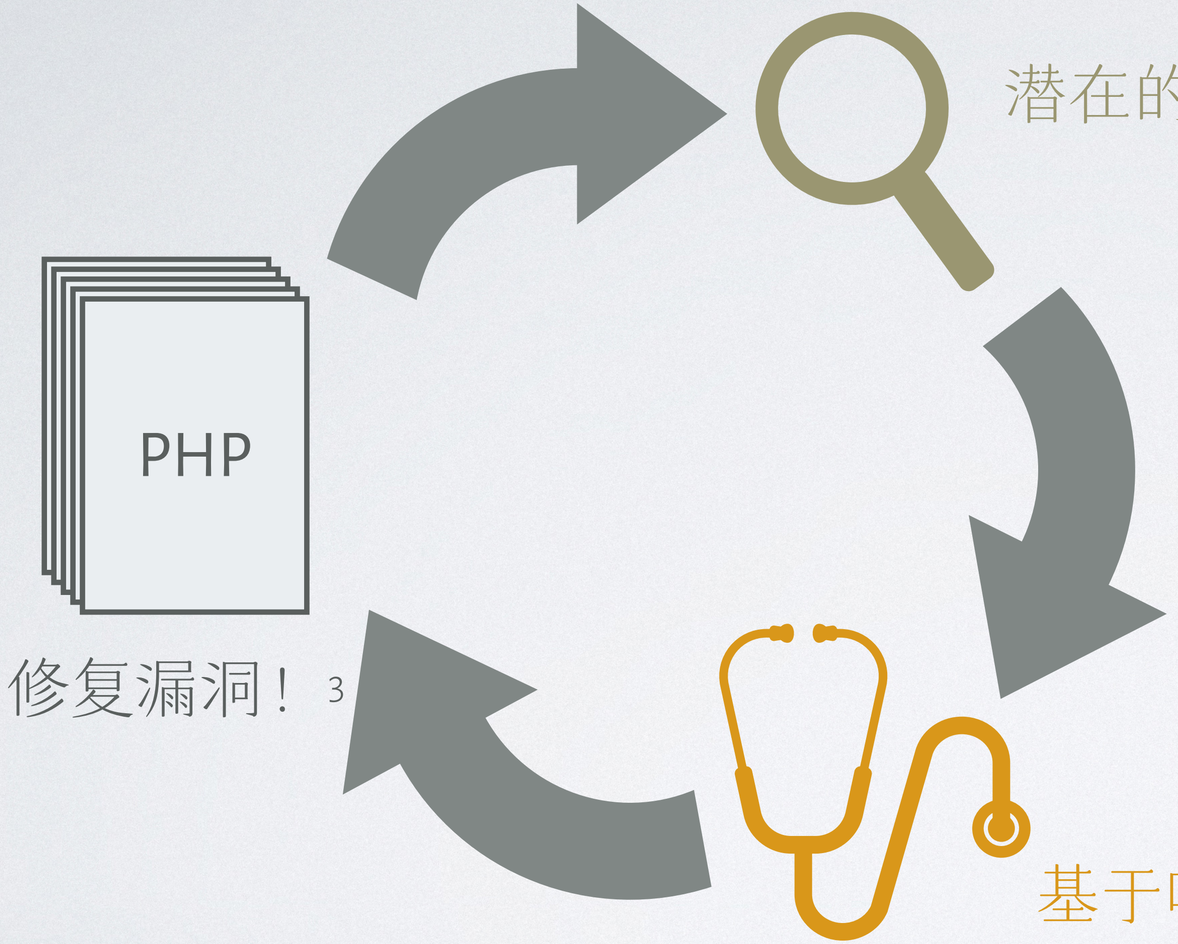


潜在的拒绝服务漏洞 [1]

基于哈希冲突的具体攻击实例 [2]

[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

发现软件的拒绝服务漏洞



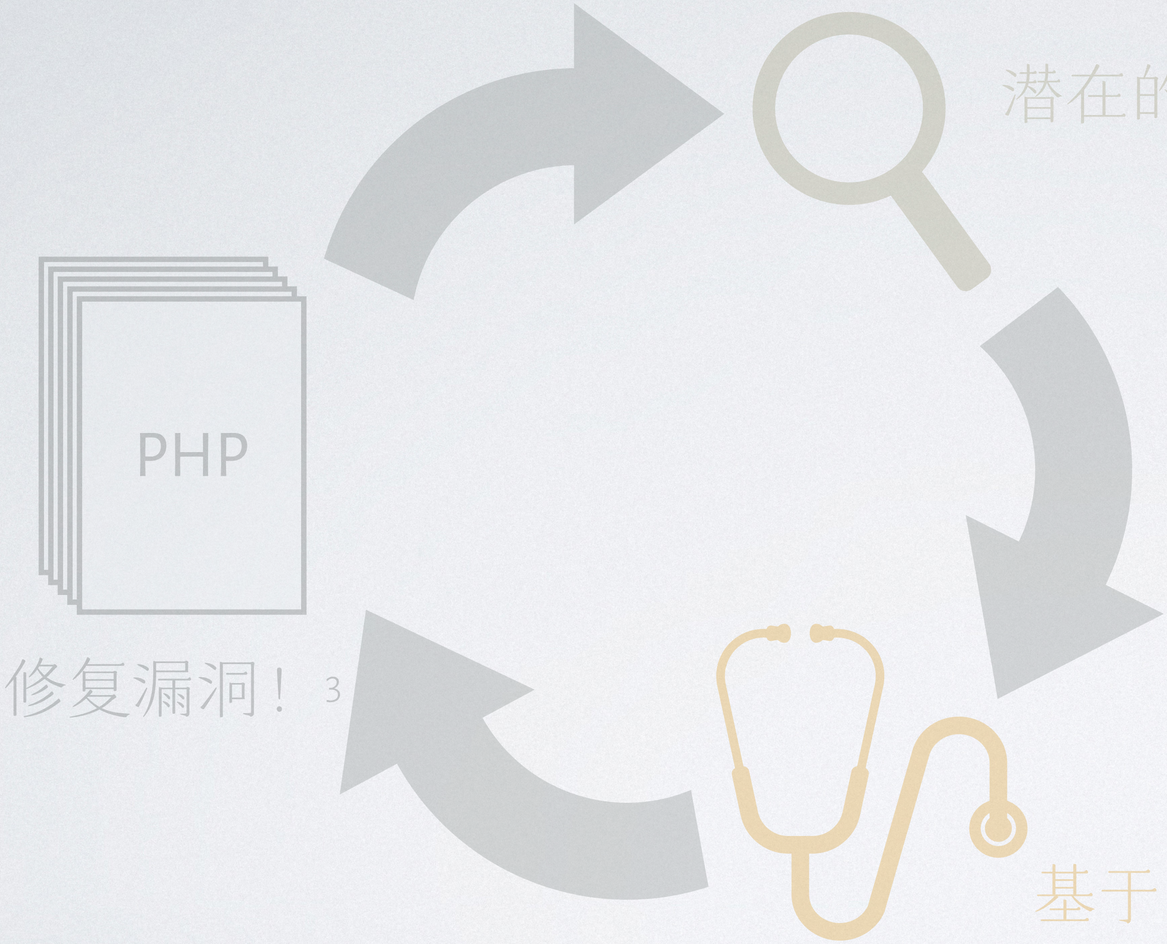
潜在的拒绝服务漏洞 [1]

基于哈希冲突的具体攻击实例 [2]

修复漏洞！ [3]

[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

发现软件的拒绝服务漏洞



潜在的拒绝服务漏洞 [1]

修复漏洞！ [3]

基于哈希冲突的具体攻击实例 [2]

一类拒绝服务漏洞与程序的资源消耗相关：比如不良哈希函数导致时间复杂度从线性退化到平方

[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

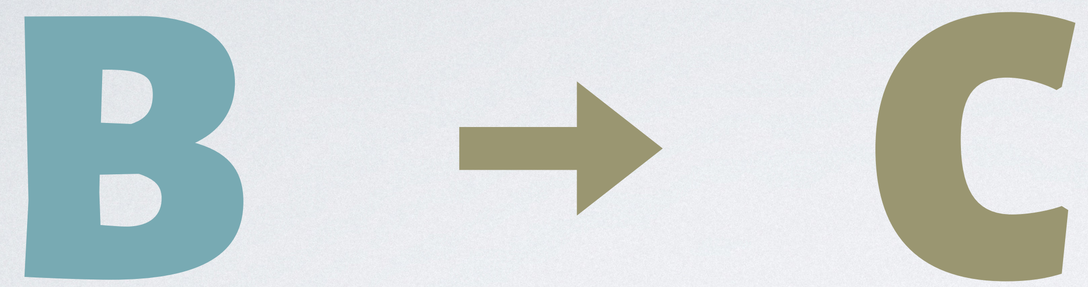
系统编程

系统编程

B

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- 无类型
- 非常慢！

系统编程



- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- 无类型
- 非常慢！

- 1972 年
- 贝尔实验室
- 通用编程语言
  - 长期被用来开发操作系统、设备驱动、协议栈等
- 静态类型系统
- 很快啊！

系统编程

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- 无类型
- 非常慢！

- 1972 年
- 贝尔实验室
- 通用编程语言
  - 长期被用来开发操作系统、设备驱动、协议栈等
- 静态类型系统
- 很快啊！

- 2015 年
- Mozilla
- 通用编程语言
  - 强调性能、安全以及并发
- 基于类型的内存安全
- 性能还不错！

系统编程

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- 无类型
- 非常慢！

- 1972 年
- 贝尔实验室
- 通用编程语言
  - 长期被用来开发操作系统、设备驱动、协议栈等
- 静态类型系统
- 很快啊！

- 2015 年
- Mozilla
- 通用编程语言
  - 强调性能、安全以及并发
- 基于类型的内存安全
- 性能还不错！

系统编程

- 1969 年
- 贝尔实验室
- 为系统和编译器开发设计
- 无类型
- 非常慢！

- 1972 年
- 贝尔实验室
- 通用编程语言
  - 长期被用来开发操作系统、设备驱动、协议栈等
- 静态类型系统
- 很快啊！

- 2015 年
- Mozilla
- 通用编程语言
  - 强调性能、安全以及并发
- 基于类型的内存安全
- 性能还不错！

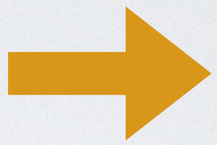
安全的系统编程

无安全保障

非常慢

类型安全

很快啊



类型、内存、并发安全

挺快的

安全的系统编程

无安全保障

非常慢

类型安全

很快啊

类型、内存、并发安全

挺快的

- 系统编程语言不仅要提供**安全保障**，还要提供**性能保障**

安全的系统编程

无安全保障

非常慢

类型安全

很快啊

类型、内存、并发安全

挺快的

- 系统编程语言不仅要提供**安全保障**，还要提供**性能保障**
- “资源安全”包含三方面：

安全的系统编程

无安全保障

非常慢

类型安全

很快啊

类型、内存、并发安全

挺快的

- 系统编程语言不仅要提供**安全保障**，还要提供**性能保障**
- “资源安全”包含三方面：

算法复杂度符合设计

安全的系统编程

无安全保障

非常慢

类型安全

很快啊

类型、内存、并发安全

挺快的

- 系统编程语言不仅要提供**安全保障**，还要提供**性能保障**
- “资源安全”包含三方面：

算法复杂度符合设计 **物理资源消耗满足预期**

安全的系统编程

无安全保障

非常慢

类型安全

很快啊

类型、内存、并发安全

挺快的

- 系统编程语言不仅要提供**安全保障**，还要提供**性能保障**
- “资源安全”包含三方面：

算法复杂度符合设计 | **物理资源消耗满足预期** | **没有资源相关的安全漏洞**

Rust

从资源的视角看内存

5 分钟学 Rust

5 分钟学 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
}
```

5 分钟学 Rust

```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
```

5 分钟学 Rust

```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;
```

5 分钟学 Rust

```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域
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}
let two = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;
```

5 分钟学 Rust

```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域
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}
let two = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;
```

```rust
// 模式匹配
match two {
    OptionalI32::AnI32(n) => println!("i32: {}", n),
    OptionalI32::Nothing => println!("nothing"),
}
```

5 分钟学 Rust

5 分钟学 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;
```

5 分钟学 Rust

```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); // 这个会报错
```

5 分钟学 Rust

```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); // 这个会报错
```

```rust
// 使用Box定义递归类型
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};
```

5 分钟学 Rust

```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); // 这个会报错
```

```rust
// 使用Box定义递归类型
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拥有这个列表的所有权
let l1 = Cons(1, Box::new(Cons(2, Box::new(Nil))));
// 列表的所有权从l1转移到l2
let l2 = l1;
// let l3 = l1; // 这个会报错
```

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

- 对 `tl: Box<List>` 解引用后，`tl` 指向的内存会被自动释放

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

- 对 `tl: Box<List>` 解引用后，`tl` 指向的内存会被自动释放
- 为了构建新的 `List`，需要申请一块内存来存放数据

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

- 对 tl: Box<List> 解引用后，tl 指向的内存会被自动释放
- 为了构建新的 List，需要申请一块内存来存放数据

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice(l: List) -> List {
    match l {
        Nil => Nil,
        Cons(hd, tl) => Cons(hd, Box::new(Cons(hd, Box::new(twice(*tl))))),
    }
}
```

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

- 对 tl: Box<List> 解引用后，tl 指向的内存会被自动释放
- 为了构建新的 List，需要申请一块内存来存放数据

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice(l: List) -> List {
    match l {
        Nil => Nil,
        Cons(hd, tl) => Cons(hd, Box::new(Cons(hd, Box::new(twice(*tl))))),
    }
}
```

在 Rust 中进行函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

- 对 tl: Box<List> 解引用后，tl 指向的内存会被自动释放
- 为了构建新的 List，需要申请一块内存来存放数据

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice(l: List) -> List {
    match l {
        Nil => Nil,
        Cons(hd, tl) => Cons(hd, Box::new(Cons(hd, Box::new(twice(*tl))))),
    }
}
```

在 Rust 中进行更高效的函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

这是一个 Box<List>

在 Rust 中进行更高效的函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

这是一个 Box<List>

将 Δ 指向内存中的列表
转移到 tl 变量所拥有

在 Rust 中进行更高效的函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

这是一个 Box<List>

将 Δ 指向内存中的列表
转移到 tl 变量所拥有

复用 Δ 指向的内存来存储
（并拥有）递归调用返回的列表

在 Rust 中进行更高效的函数式编程

```
// 计算列表l1和l2连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

这是一个 Box<List>

将 Δ 指向内存中的列表
转移到 tl 变量所拥有

复用 Δ 指向的内存来存储
（并拥有）递归调用返回的列表

通过复用 Box<List> 完成了函数式编程，避免了额外的内存开销

在 Rust 中进行更高效的函数式编程

```
// 计算将a插入有序列表l中后所得的列表
fn insert(a: i32, l: List) -> List {
    match l {
        Nil => Cons(a, Box::new(Nil)),
        Cons(hd, tl) => {
            if a <= hd {
                Cons(a, Box::new(Cons(hd, tl)))
            } else {
                Cons(hd, Box::new(insert(a, *tl)))
            }
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
// 计算将a插入有序列表l中后所得的列表
fn insert(a: i32, l: List) -> List {
    match l {
        Nil => Cons(a, Box::new(Nil)),
        Cons(hd, tl) => {
            if a <= hd {
                Cons(a, Box::new(Cons(hd, tl)))
            } else {
                Cons(hd, Box::new(insert(a, *tl)))
            }
        }
    }
}
```

因为返回列表长度加一，所以一定需要一个额外的 Box::new

在 Rust 中进行更高效的函数式编程

```rust
// 计算将a插入有序列表l中后所得的列表
fn insert2(mut Δ1: Box<List>, a: i32, l: List) -> List {
    match l {
        Nil => {
            *Δ1 = Nil;
            Cons(a, Δ1)
        }
        Cons(hd, mut Δ2) => {
            let tl = *Δ2;
            if a <= hd {
                *Δ2 = tl;
                *Δ1 = Cons(hd, Δ2);
                Cons(a, Δ1)
            } else {
                *Δ2 = insert2(Δ1, a, tl);
                Cons(hd, Δ2)
            }
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
// 计算将a插入有序列表l中后所得的列表
fn insert2(mut Δ1: Box<List>, a: i32, l: List) -> List {
    match l {
        Nil => {
            *Δ1 = Nil;
            Cons(a, Δ1)
        }
        Cons(hd, mut Δ2) => {
            let tl = *Δ2;
            if a <= hd {
                *Δ2 = tl;
                *Δ1 = Cons(hd, Δ2);
                Cons(a, Δ1)
            } else {
                *Δ2 = insert2(Δ1, a, tl);
                Cons(hd, Δ2)
            }
        }
    }
}
```

显式传入一块可用的内存，在插入 a 的位置使用

在 Rust 中进行更高效的函数式编程

```
// 计算将a插入有序列表l中后所得的列表
fn insert2(mut Δ1: Box<List>, a: i32, l: List) -> List {
    match l {
        Nil => {
            *Δ1 = Nil;
            Cons(a, Δ1)
        }
        Cons(hd, mut Δ2) => {
            let tl = *Δ2;
            if a <= hd {
                *Δ2 = tl;
                *Δ1 = Cons(hd, Δ2);
                Cons(a, Δ1)
            } else {
                *Δ2 = insert2(Δ1, a, tl);
                Cons(hd, Δ2)
            }
        }
    }
}
```

显式传入一块可用的内存，在插入 a 的位置使用

当需要递归调用时，把这块可用内存也作为参数传过去

在 Rust 中进行更高效的函数式编程

```
// 计算将a插入有序列表l中后所得的列表
fn insert2(mut Δ1: Box<List>, a: i32, l: List) -> List {
    match l {
        Nil => {
            *Δ1 = Nil;
            Cons(a, Δ1)
        }
        Cons(hd, mut Δ2) => {
            let tl = *Δ2;
            if a <= hd {
                *Δ2 = tl;
                *Δ1 = Cons(hd, Δ2);
                Cons(a, Δ1)
            } else {
                *Δ2 = insert2(Δ1, a, tl);
                Cons(hd, Δ2)
            }
        }
    }
}
```

显式传入一块可用的内存，在插入 a 的位置使用

当需要递归调用时，把这块可用内存也作为参数传过去

通过显式传递函数计算中所需的额外内存，避免了在函数内部进行内存分配

在 Rust 中进行更高效的函数式编程

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

```
// 计算将a插入有序列表l中后所得的列表
fn insert3(Δ1: Box<List>, a: i32, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => cons(Δ1, a, nil()),
        ListDestr::Cons(Δ2, hd, tl) => {
            if a <= hd {
                cons(Δ1, a, cons(Δ2, hd, tl))
            } else {
                cons(Δ2, hd, insert3(Δ1, a, tl))
            }
        }
    }
}
```

在 Rust 中进行更高效的函数式编程

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

```
// 计算将a插入有序列表l中后所得的列表
fn insert3(Δ1: Box<List>, a: i32, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => cons(Δ1, a, nil()),
        ListDestr::Cons(Δ2, hd, tl) => {
            if a <= hd {
                cons(Δ1, a, cons(Δ2, hd, tl))
            } else {
                cons(Δ2, hd, insert3(Δ1, a, tl))
            }
        }
    }
}
```

函数式但是“原地”更新数据结构！

Non-Size-Increasing Computation [1]

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 **EXPTIME**，即指数时间可计算

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 **EXPTIME**，即指数时间可计算

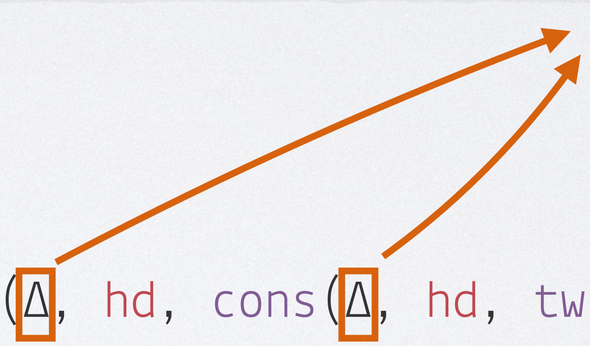
```
// 计算将l中每个元素拷贝两次所得的列表
fn twice2(l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ, hd, tl) => cons(Δ, hd, cons(Δ, hd, twice2(tl))),
    }
}
```

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 **EXPTIME**，即指数时间可计算

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice2(l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ, hd, tl) => cons(Δ, hd, cons(Δ, hd, twice2(tl))),
    }
}
```



Δ 只能被使用一次，故该程序**不能通过类型检查**

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 **EXPTIME**，即指数时间可计算

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice2(mut extra: Vec<Box<List>>, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ1, hd, tl) => {
            let Δ2 = extra.pop().unwrap();
            cons(Δ1, hd, cons(Δ2, hd, twice2(extra, tl)))
        }
    }
}
```

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

Non-Size-Increasing Computation [1]

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 **EXPTIME**，即指数时间可计算

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice2(mut extra: Vec<Box<List>>, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ1, hd, tl) => {
            let Δ2 = extra.pop().unwrap();
            cons(Δ1, hd, cons(Δ2, hd, twice2(extra, tl)))
        }
    }
}
```

twice2 函数需要的额外内存块等于 1 的长度，不是一个常数，故通过一个 Vec 传进来

[1] M. Hofmann. 2000. A type system for bounded space and functional in-place update. In *ESOP'00*.

从资源的视角看内存

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个**内存块**

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个**内存块**

在构建 Cons 列表时，需要一单位**资源**

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个**内存块**

在构建 Cons 列表时，需要一单位**资源**

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个内存块

在构建 Cons 列表时，需要一单位资源

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

在解构 Cons 列表时，得到一个可用内存块

从资源的视角看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>, hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个内存块

在构建 Cons 列表时，需要一单位资源

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

在解构 Cons 列表时，得到一个可用内存块

在解构 Cons 列表时，得到一单位可用资源

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

某种**资源**

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

某种**资源**

将**资源**存储在数据结构中

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

某种**资源**

将**资源**存储在数据结构中

从数据结构中获取**资源**

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数

以一个抽象的视角来看

```
// Trait就像是Interface或Typeclass
trait ListTrait: Sized {
    type Resource;

    fn nil() -> Self;
    fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

某种**资源**

将**资源**存储在数据结构中

从数据结构中获取**资源**

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数
- 若将资源视作数据结构**类型的静态信息**，则得到了一种**静态资源分析**方法

以一个抽象的视角来看

```
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

以一个抽象的视角来看

```
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

以一个抽象的视角来看

```
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```
// 资源的计数单位
struct Delta;
```

以一个抽象的视角来看

```
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```
// 资源的计数单位
struct Delta;
```

在类型层面，通过 `(Delta, Delta, …)` 来表示多个单位的资源

以一个抽象的视角来看

```rust
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```rust
// 资源的计数单位
struct Delta;
```

在类型层面，通过 `(Delta, Delta, …)` 来表示多个单位的资源

```rust
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

以一个抽象的视角来看

```rust
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```rust
// 资源的计数单位
struct Delta;
```

在类型层面，通过 `(Delta, Delta, …)` 来表示多个单位的资源

```rust
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

输入列表中每个元素带 2 单位的资源

以一个抽象的视角来看

```rust
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```rust
// 资源的计数单位
struct Delta;
```

在类型层面，通过 `(Delta, Delta, …)` 来表示多个单位的资源

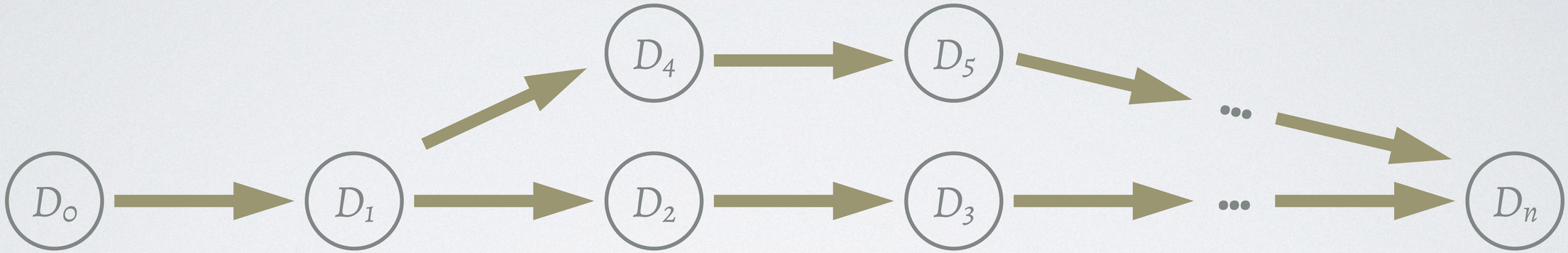
```rust
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

输入列表中每个元素带 2 单位的资源

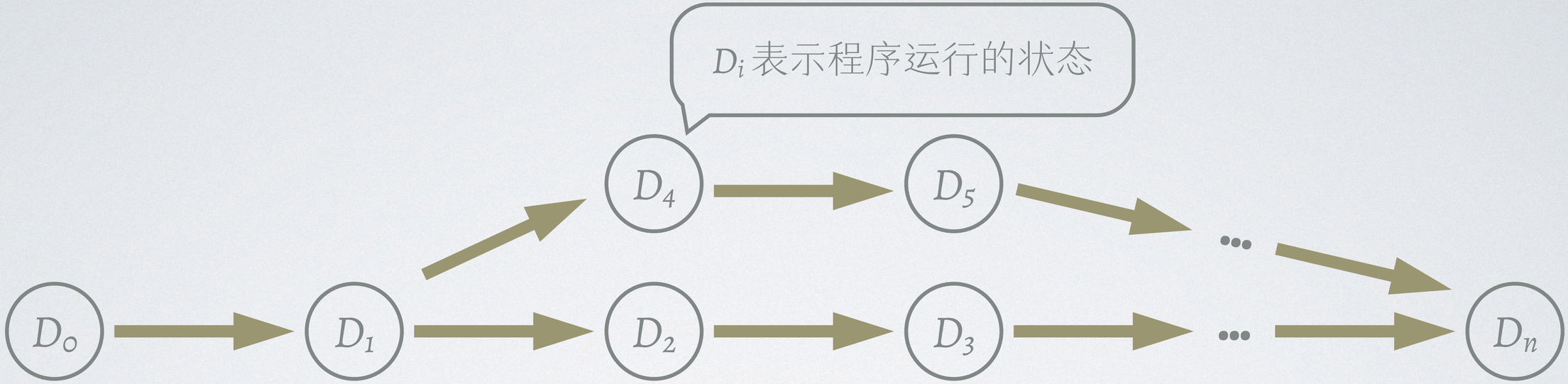
输出列表中每个元素带 1 单位的资源

数据结构存储的资源即其势能

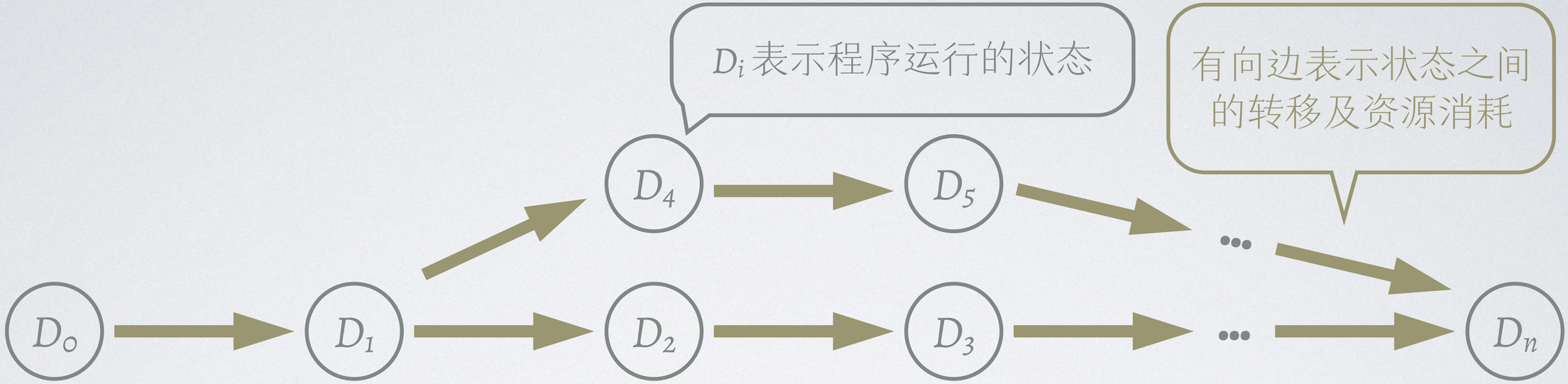
数据结构存储的资源即其势能



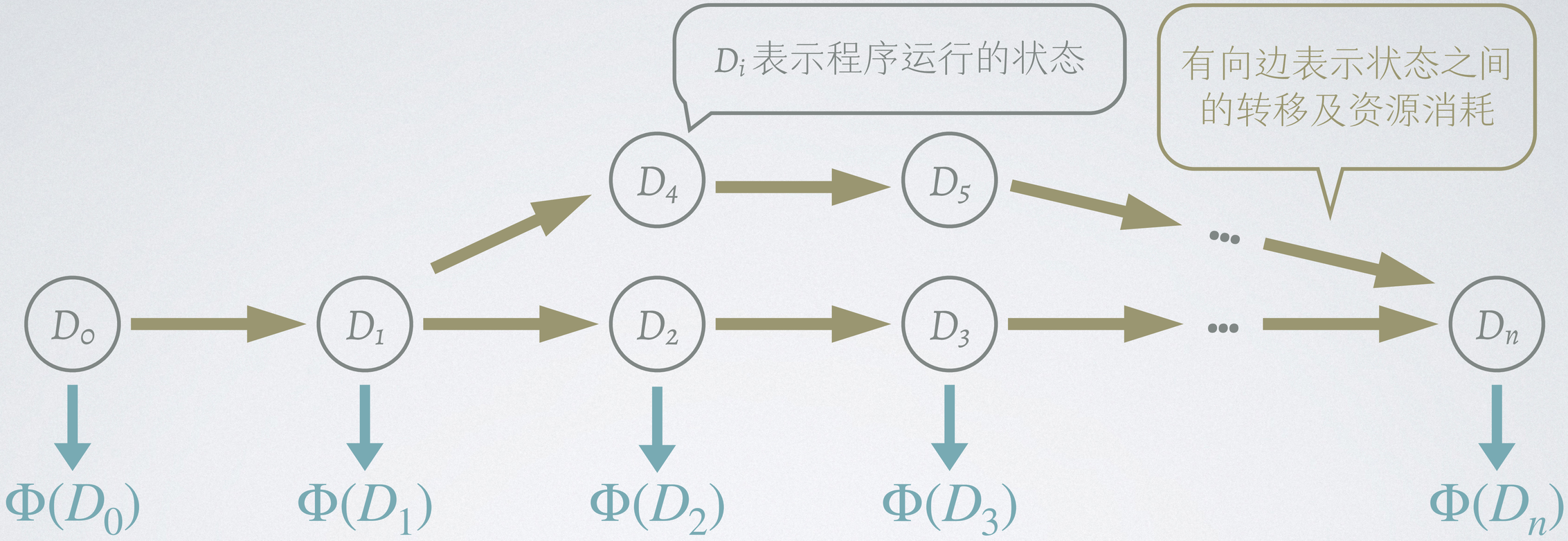
数据结构存储的资源即其势能



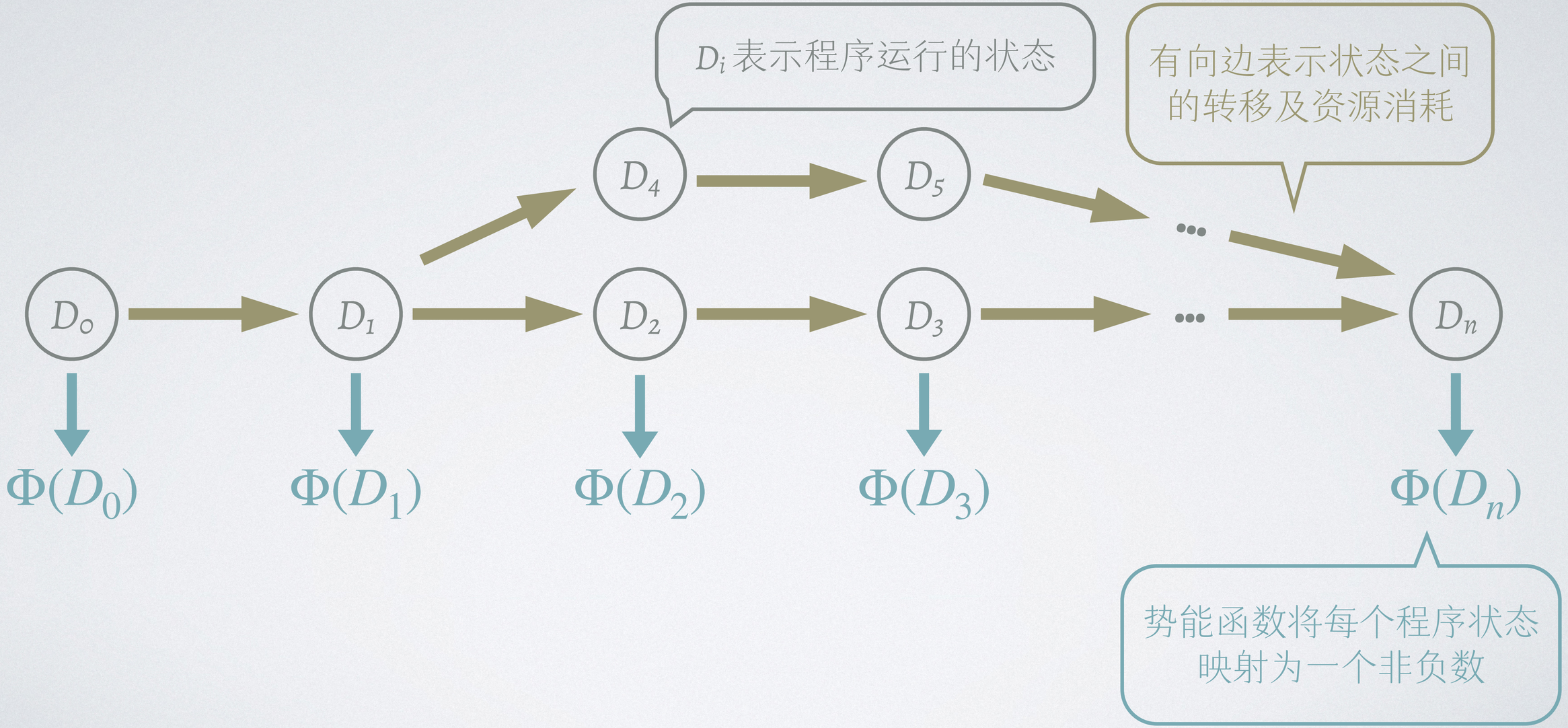
数据结构存储的资源即其势能



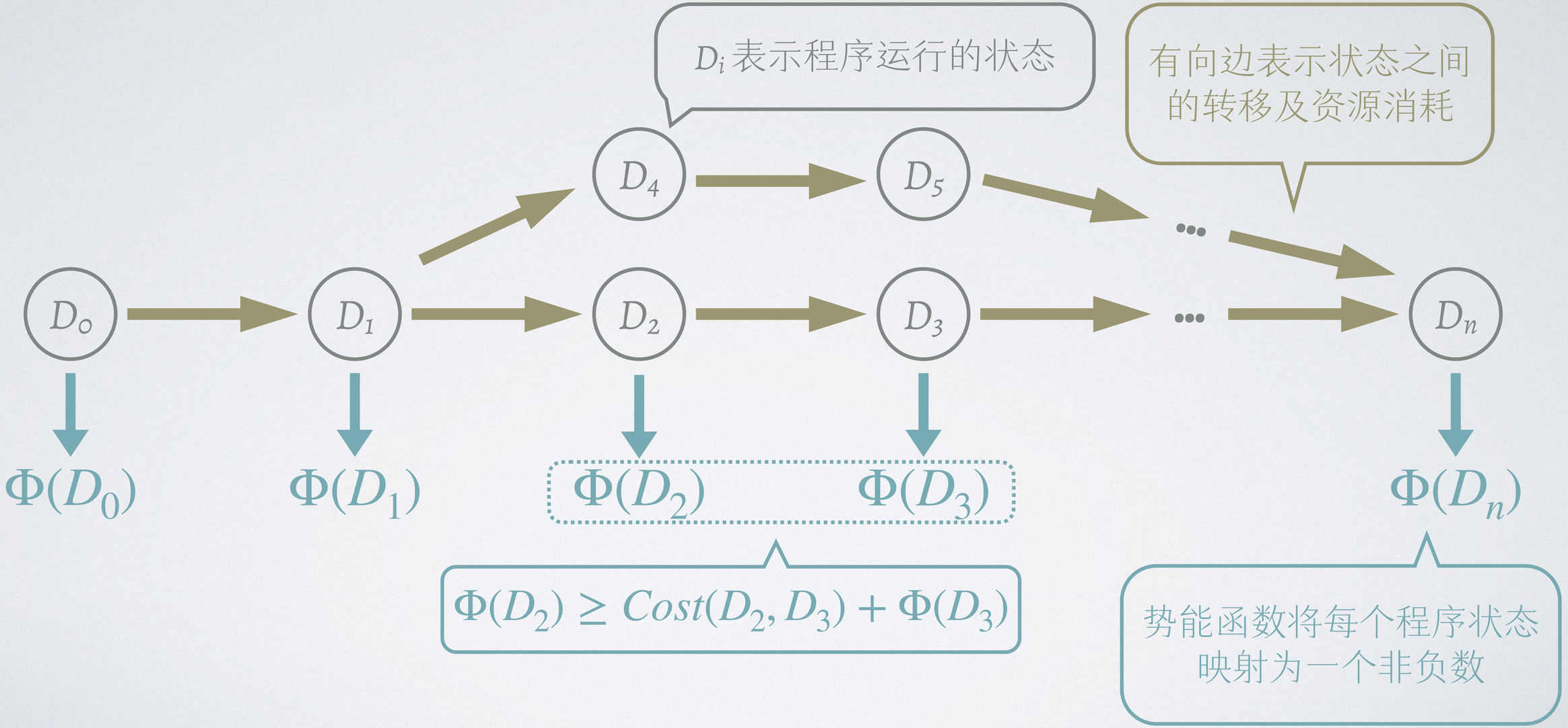
数据结构存储的资源即其势能



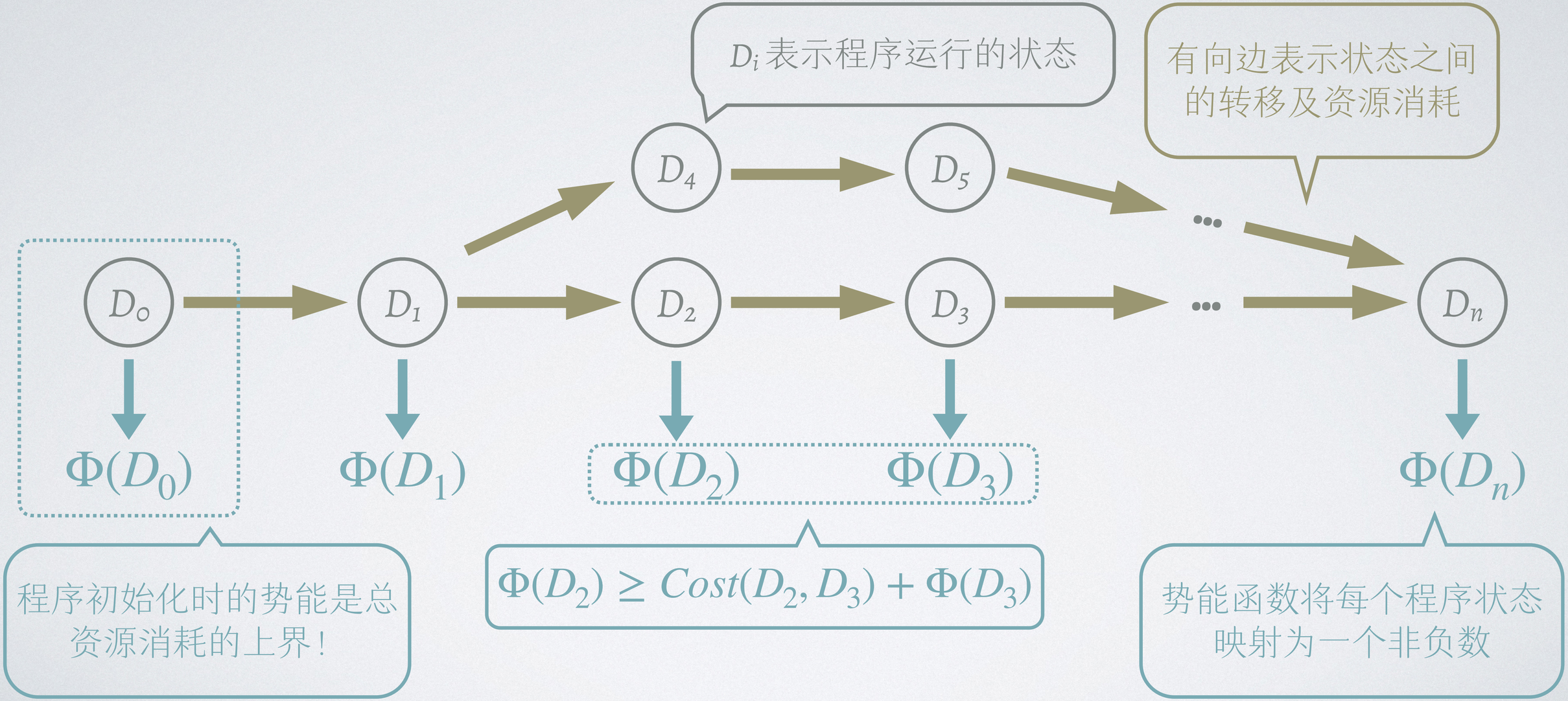
数据结构存储的资源即其势能



数据结构存储的资源即其势能

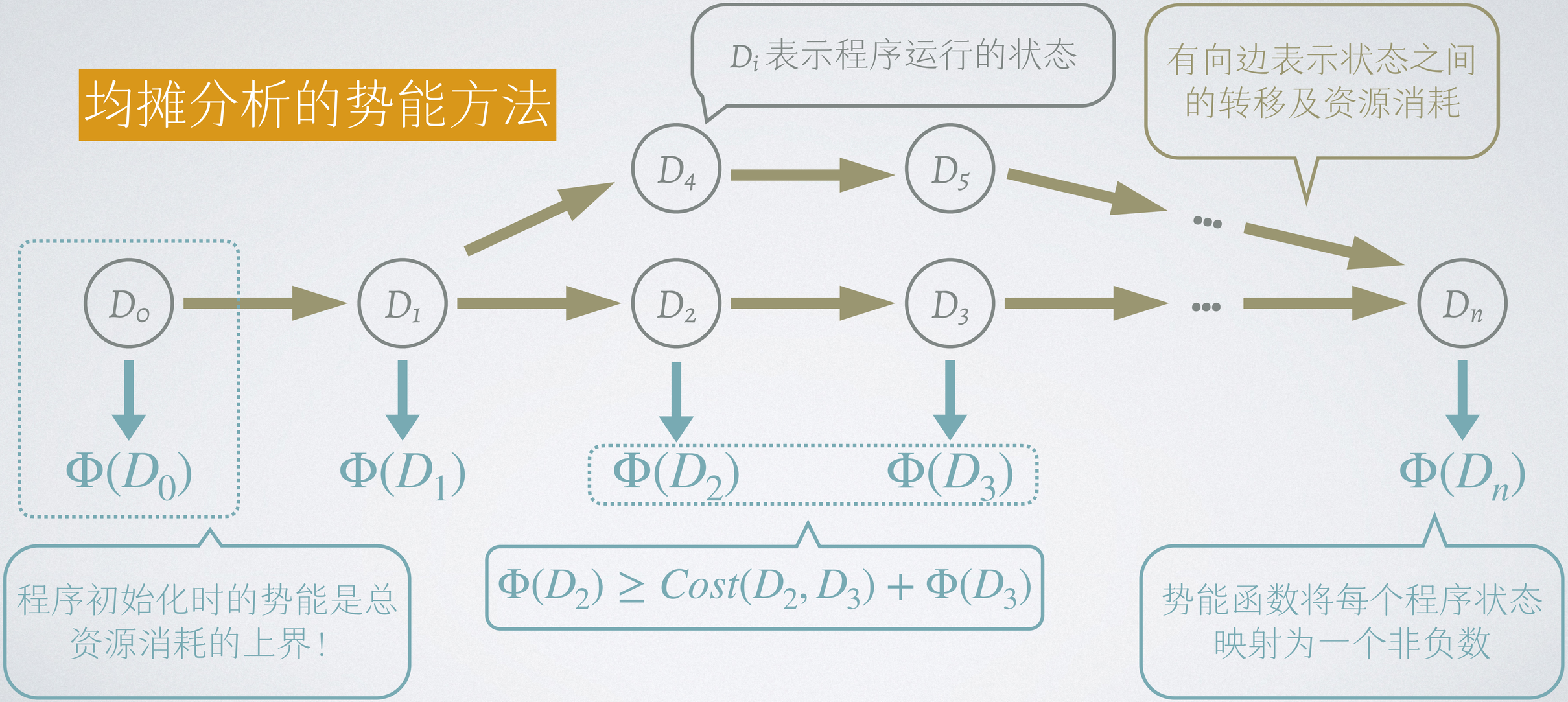


数据结构存储的资源即其势能



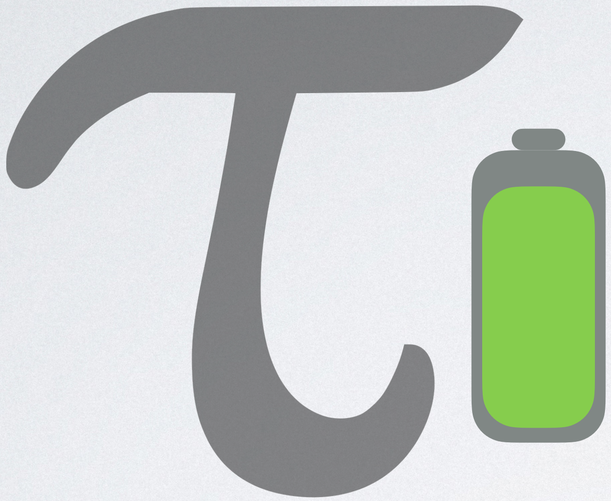
数据结构存储的资源即其势能

均摊分析的势能方法

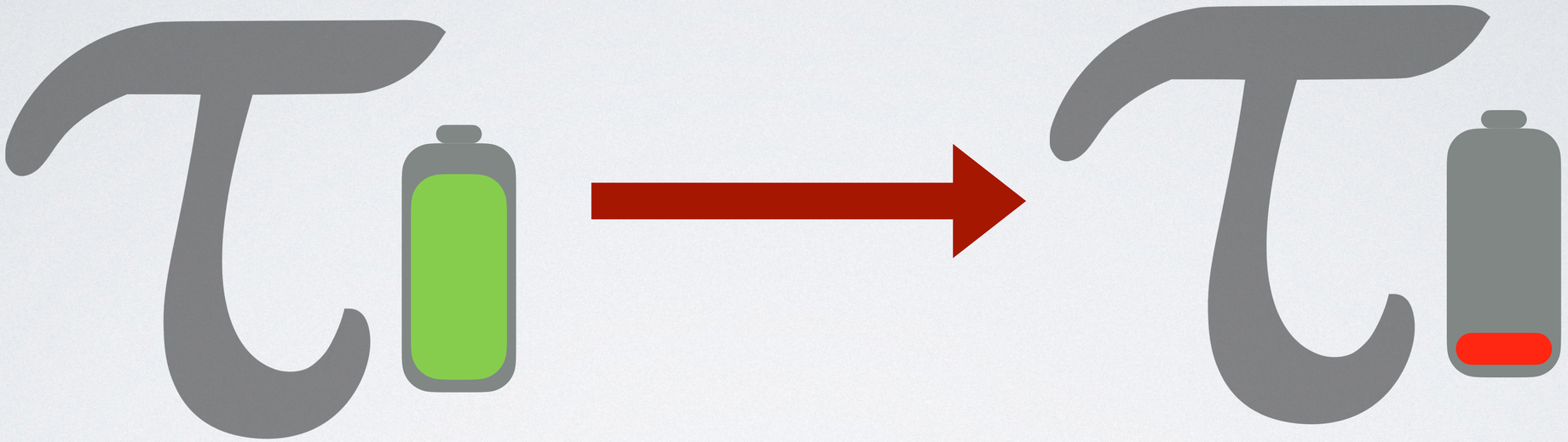


基于类型的资源消耗分析

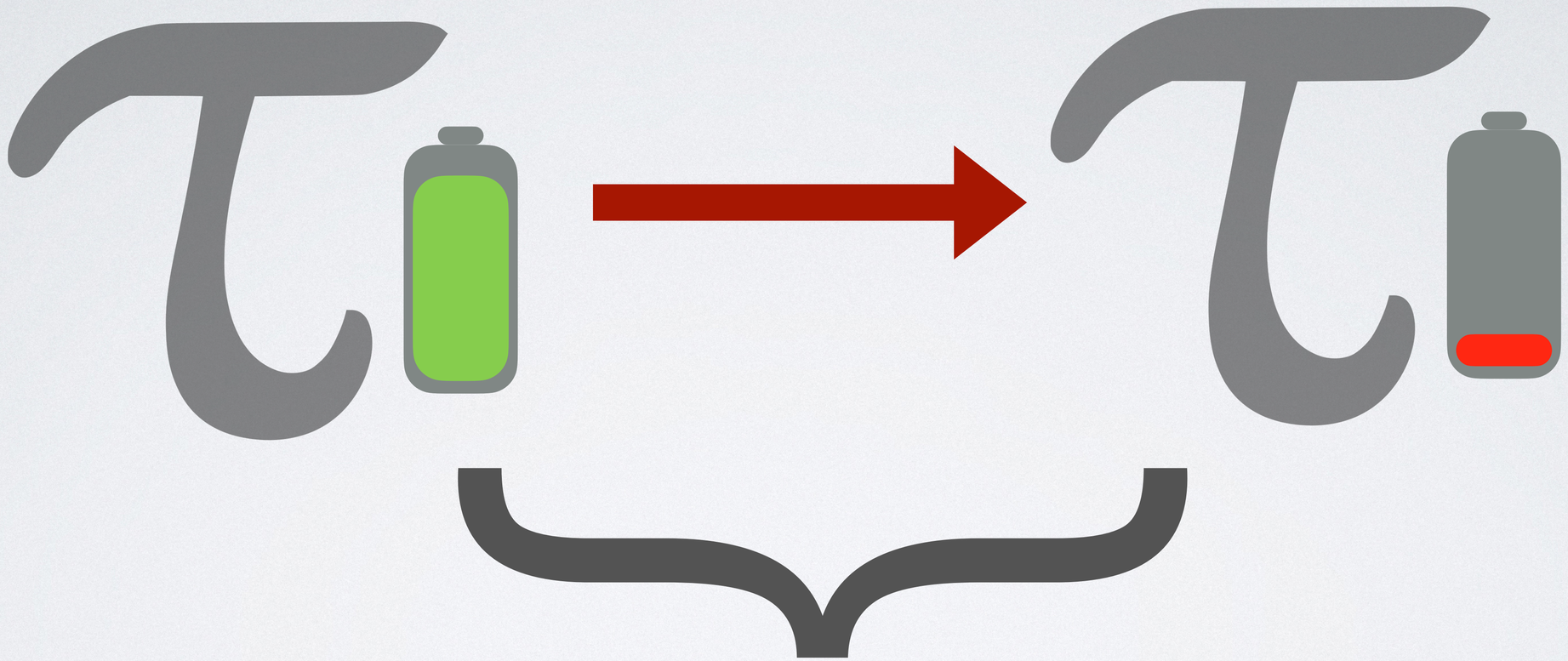
基于类型的资源消耗分析



基于类型的资源消耗分析



基于类型的资源消耗分析



资源消耗

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

表示有 2 个 Delta，即 2 单位的势能

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>, ⟶ 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>, ⟶ 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>, ⟶ 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>, ⟶ 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => T2::cons(hd, T2::cons(hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,  → 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>,  → 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => T2::cons(hd, T2::cons(hd, twice3(tl))),
    }    得到 2 单位势能
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,   → 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>,   → 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => T2::cons(hd, T2::cons(hd, twice3(tl))),
    }
}
```

存储 1 单位势能

得到 2 单位势能

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,   → 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>,   → 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => T2::cons(hd, T2::cons(hd, twice3(tl))),
    }
}
```

存储 1 单位势能

存储 1 单位势能

得到 2 单位势能

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => T2::cons(hd, T2::cons(hd, twice3(tl))),
    }
}
```

表示有 2 个 Delta，即 2 单位的势能

表示有 1 个 Delta，即 1 单位的势能

存储 1 单位势能

存储 1 单位势能

得到 2 单位势能

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

消耗 1 单位势能

- 通过插入 tick<1>() 显式将递归函数调用次数视作资源消耗

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,   → 表示有 3 个 Delta，即 3 单位的势能
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();          ← 消耗 1 单位势能
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：

  $$\Phi(\ell) = 3 \cdot |\ell|$$

- 返回时的势能函数：

  $$\Phi(\ell') = 1 \cdot |\ell'|$$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,   → 表示有 3 个 Delta，即 3 单位的势能
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {   得到 3 单位势能
            tick<1>();        ← 消耗 1 单位势能
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：

  $$\Phi(\ell) = 3 \cdot |\ell|$$

- 返回时的势能函数：

  $$\Phi(\ell') = 1 \cdot |\ell'|$$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,   → 表示有 3 个 Delta，即 3 单位的势能
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {   得到 3 单位势能
            tick<1>();        ← 消耗 1 单位势能
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }                          ↑ 存储 1 单位势能   ↑ 存储 1 单位势能
    }
}
```

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：

  $$\Phi(\ell) = 3 \cdot |\ell|$$

- 返回时的势能函数：

  $$\Phi(\ell') = 1 \cdot |\ell'|$$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 3 个 Delta，即 3 单位的势能

得到 3 单位势能

消耗 1 单位势能

存储 1 单位势能

存储 1 单位势能

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：

  $\Phi(\ell) = 3 \cdot |\ell|$
- 返回时的势能函数：

  $\Phi(\ell') = 1 \cdot |\ell'|$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 3 个 Delta，即 3 单位的势能

得到 3 单位势能

消耗 1 单位势能

存储 1 单位势能

存储 1 单位势能

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：
  $\Phi(\ell) = 3 \cdot |\ell|$
- 返回时的势能函数：
  $\Phi(\ell') = 1 \cdot |\ell'|$
- 根据函数性质，有
  $|\ell'| = 2 \cdot |\ell|$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 3 个 Delta，即 3 单位的势能

得到 3 单位势能

消耗 1 单位势能

存储 1 单位势能

存储 1 单位势能

- 通过插入 `tick<1>()` 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：
  $\Phi(\ell) = 3 \cdot |\ell|$
- 返回时的势能函数：
  $\Phi(\ell') = 1 \cdot |\ell'|$
- 根据函数性质，有
  $|\ell'| = 2 \cdot |\ell|$
- 所以资源消耗为
  $\Phi(\ell) - \Phi(\ell') = 1 \cdot |\ell|$

带有势能标注的类型

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 3 个 Delta，即 3 单位的势能

得到 3 单位势能

消耗 1 单位势能

存储 1 单位势能

存储 1 单位势能

适用于分析算法复杂度

- 通过插入 tick<1>() 显式将递归函数调用次数视作资源消耗
- 初始时的势能函数：
  $\Phi(\ell) = 3 \cdot |\ell|$
- 返回时的势能函数：
  $\Phi(\ell') = 1 \cdot |\ell'|$
- 根据函数性质，有
  $|\ell'| = 2 \cdot |\ell|$
- 所以资源消耗为
  $\Phi(\ell) - \Phi(\ell') = 1 \cdot |\ell|$

基于势能方法的自动资源分析

[HJ03]	线性形式的资源消耗上界，通过线性规划进行自动分析
[JHL+10]	高阶函数和参数多态
[Atkey10]	分离逻辑，副作用，堆操作
[HDW17]	多元多项式形式的资源消耗上界，归纳数据类型，高阶函数
[HM18]	对数形式的资源消耗上界
[WH19]	最坏情况资源消耗分析，最坏情况测试输入生成
[KWH+19]	满足资源消耗规约的程序合成
[KH20]	指数形式的资源消耗上界

[HJ03] M. Hofmann and S. Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03*.
[JHL+10] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *POPL'10*.
[Atkey10] R. Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP'10*.
[HDW17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.
[HM18] M. Hofmann and G. Moser. 2018. Analysis of Logarithmic Amortised Complexity. Available on: https://arxiv.org/abs/1807.08242.
[WH19] D. Wang and J. Hoffmann. 2019. Type-Guided Worst-Case Input Generation. In *POPL'19*.
[KWH+19] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI'19*.
[KH20] D. M. Kahn and J. Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *FoSSaCS'20*.

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

消耗 **1** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

消耗 **1** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

消耗 **1** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

$$p \geq 0$$
$$q \geq 0$$
$$p \geq 1 + q + q$$

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

消耗 **1** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

$$p \geq 0$$
$$q \geq 0$$
$$p \geq 1 + q + q$$

- $p = 3, q = 1$: `ListTraitStatic<3> -> ListTraitStatic<1>`

通过线性规划进行自动资源分析

```
// 计算将l中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

得到 **p** 单位势能

消耗 **1** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

$$p \geq 0$$
$$q \geq 0$$
$$p \geq 1 + q + q$$

- $p = 3, q = 1$: `ListTraitStatic<3> -> ListTraitStatic<1>`
- $p = 5, q = 2$: `ListTraitStatic<5> -> ListTraitStatic<2>`

源程序资源消耗复杂度分析

源程序资源消耗复杂度分析

- 在软件源程序层面，对某种抽象的资源度量，进行资源消耗复杂度分析
  - 例如，以**循环和递归调用的次数**为度量，进行的就是**时间复杂度**分析

源程序资源消耗复杂度分析

- 在软件源程序层面，对某种抽象的资源度量，进行资源消耗复杂度分析
  - 例如，以**循环和递归调用的次数**为度量，进行的就是**时间复杂度**分析

- **研究问题分两方面：**
  - 在分析中考量语言本身的**内存抽象**、**并发抽象**和**抽象机制被破坏**的情况
  - 从新语言的视角看，如何设计语言使得资源分析更容易、更精确？

源程序资源消耗复杂度分析

- 在软件源程序层面，对某种抽象的资源度量，进行资源消耗复杂度分析
  - 例如，以**循环和递归调用的次数**为度量，进行的就是**时间复杂度**分析

- **研究问题分两方面：**

  以 Rust 为例

  - 在分析中考量语言本身的**内存抽象**、**并发抽象**和**抽象机制被破坏**的情况
  - 从新语言的视角看，如何设计语言使得资源分析更容易、更精确？

进行中工作：通过转译分析 Rust 程序的资源消耗

[1] S. Ho and J. Protzenko. 2022. AENEAS: Rust Verification by Functional Translation. In *ICFP'22*.

[2] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

进行中工作：通过转译分析 Rust 程序的资源消耗


```
fn list_consume(l: &mut List) {
    match l {
        Nil => {}
        Cons(_, tl) => {
            tick<1>();
            list_consume(tl);
        }
    }
    *l = Nil;
}

fn list_consume_twice(l: &mut List) {
    list_consume(l);
    list_consume(l);
}
```

[1] S. Ho and J. Protzenko. 2022. AENEAS: Rust Verification by Functional Translation. In *ICFP'22*.

[2] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

进行中工作：通过转译分析 Rust 程序的资源消耗


```rust
fn list_consume(l: &mut List) {
    match l {
        Nil => {}
        Cons(_, tl) => {
            tick<1>();
            list_consume(tl);
        }
    }
    *l = Nil;
}

fn list_consume_twice(l: &mut List) {
    list_consume(l);
    list_consume(l);
}
```

```
let rec list_consume_fwd_back l =
  match l with
  | [] -> []
  | _ :: tl ->
    let _ = Raml.tick 1.0 in
    let _ = list_consume_fwd_back tl in
    []

let list_consume_twice_fwd_back l =
  let l0 = list_consume_fwd_back l in
  list_consume_fwd_back l0
```

[1] S. Ho and J. Protzenko. 2022. AENEAS: Rust Verification by Functional Translation. In *ICFP'22*.

[2] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

进行中工作：通过转译分析 Rust 程序的资源消耗


```rust
fn list_consume(l: &mut List) {
    match l {
        Nil => {}
        Cons(_, tl) => {
            tick<1>();
            list_consume(tl);
        }
    }
    *l = Nil;
}

fn list_consume_twice(l: &mut List) {
    list_consume(l);
    list_consume(l);
}
```

```
let rec list_consume_fwd_back l =
  match l with
  | [] -> []
  | _ :: tl ->
    let _ = Raml.tick 1.0 in
    let _ = list_consume_fwd_back tl in
    []

let list_consume_twice_fwd_back l =
  let l0 = list_consume_fwd_back l in
  list_consume_fwd_back l0
```

```
== list_consume_twice_fwd_back :

  Simplified bound:
     1*M
   where
     M is the number of ::-nodes of the argument
```

[1] S. Ho and J. Protzenko. 2022. AENEAS: Rust Verification by Functional Translation. In *ICFP'22*.

[2] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

进行中工作：通过转译分析 Rust 程序的资源消耗


```
fn list_consume(l: &mut List) {
    match l {
        Nil => {}
        Cons(_, tl) => {
            tick<1>();
            list_consume(tl);
        }
    }
    *l = Nil;
}

fn list_consume_twice(l: &mut List) {
    list_consume(l);
    list_consume(l);
}
```

```
let rec list_consume_fwd_back l =
  match l with
  | [] -> []
  | _ :: tl ->
    let _ = Raml.tick 1.0 in
    let _ = list_consume_fwd_back tl in
    []

let list_consume_twice_fwd_back l =
  let l0 = list_consume_fwd_back l in
  list_consume_fwd_back l0
```

```
== list_consume_twice_fwd_back :

 Simplified bound:
    1*M
  where
    M is the number of ::-nodes of the argument
```

- 这种方法的不足：
  - 对 Rust 特性覆盖不全面
  - 依赖 RAML 的分析能力
  - 不支持并发等场景

[1] S. Ho and J. Protzenko. 2022. AENEAS: Rust Verification by Functional Translation. In *ICFP'22*.

[2] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

Rust 程序的资源分析：内存抽象

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析
- 然而，引用和借用使得资源分析变得困难

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析
- 然而，引用和借用使得资源分析变得困难

```
let mut var = 4;
var = 3;
// 通过引用来暂时从一个值的所有者那里借用该值
// 在借用期间，该值不能被移动走或被修改
let ref_var: &i32 = &var;
println!("{}", *ref_var);
// 这里我们修改了var，这表明ref_var的借用结束了
var = 2;
```

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析
- 然而，引用和借用使得资源分析变得困难

```rust
let mut var = 4;
var = 3;
// 通过引用来暂时从一个值的所有者那里借用该值
// 在借用期间，该值不能被移动走或被修改
let ref_var: &i32 = &var;
println!("{}", *ref_var);
// 这里我们修改了var，这表明ref_var的借用结束了
var = 2;
```

```rust
// 通过可变引用不仅可以借用一个值，也允许
// 在借用期间，通过该可变引用对值进行修改
let ref_var2: &mut i32 = &mut var;
*ref_var2 += 3;
// let ref_var3 = &var; // 这个会报错
// var = 1; // 这个也会报错
*ref_var2 += 1;
```

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析
- 然而，引用和借用使得资源分析变得困难

```rust
let mut var = 4;
var = 3;
// 通过引用来暂时从一个值的所有者那里借用该值
// 在借用期间，该值不能被移动走或被修改
let ref_var: &i32 = &var;
println!("{}", *ref_var);
// 这里我们修改了var，这表明ref_var的借用结束了
var = 2;
```

```rust
// 通过可变引用不仅可以借用一个值，也允许
// 在借用期间，通过该可变引用对值进行修改
let ref_var2: &mut i32 = &mut var;
*ref_var2 += 3;
// let ref_var3 = &var; // 这个会报错
// var = 1; // 这个也会报错
*ref_var2 += 1;
```

- 对一个值，在某时刻，可以有一个可变引用，或者有多个不可变引用

Rust 程序的资源分析：内存抽象

- 所有权和移动语义使得我们能比较自然地利用 Rust 的类型系统进行资源分析
- 然而，引用和借用使得资源分析变得困难

```rust
let mut var = 4;
var = 3;
// 通过引用来暂时从一个值的所有者那里借用该值
// 在借用期间，该值不能被移动走或被修改
let ref_var: &i32 = &var;
println!("{}", *ref_var);
// 这里我们修改了var，这表明ref_var的借用结束了
var = 2;
```

```rust
// 通过可变引用不仅可以借用一个值，也允许
// 在借用期间，通过该可变引用对值进行修改
let ref_var2: &mut i32 = &mut var;
*ref_var2 += 3;
// let ref_var3 = &var; // 这个会报错
// var = 1; // 这个也会报错
*ref_var2 += 1;
```

- 对一个值，在某时刻，可以有一个可变引用，或者有多个不可变引用
- **问题：消耗资源是一种修改，而通过不可变引用需要允许资源消耗**

Rust 程序的资源分析：内存抽象

```rust
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

Rust 程序的资源分析：内存抽象

```
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

Rust 程序的资源分析：内存抽象

```
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

表示有 **1** 个 Delta，即 **1** 单位的势能

Rust 程序的资源分析：内存抽象

```
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

表示有 **1** 个 Delta，即 **1** 单位的势能

这里势能 Δ 的类型为 &Delta，是对一单位势能的不可变引用

Rust 程序的资源分析：内存抽象

```rust
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

表示有 **1** 个 Delta，即 **1** 单位的势能

这里势能 Δ 的类型为 &Delta，是对一单位势能的不可变引用

消耗势能的函数 tick 要求移动走 *Δ，**这并不能做到，因为只有不可变引用的借用权限**

Rust 程序的资源分析：内存抽象

```rust
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

表示有 **1** 个 Delta，即 **1** 单位的势能

这里势能 Δ 的类型为 &Delta，是对一单位势能的不可变引用

消耗势能的函数 tick 要求移动走 *Δ，**这并不能做到，因为只有不可变引用的借用权限**

- 势能是一种特殊的值，它应当可以通过不可变引用被移动或被消耗

Rust 程序的资源分析：内存抽象

```rust
// 遍历并打印l中的元素
fn print_list<T>(l: &T)
where
    T: ListTraitStatic<Delta>,
{
    match l.destr_ref() {
        None => {}
        Some((Δ, hd, tl)) => {
            tick(*Δ);
            println!("{}", *hd);
            print_list(tl);
        }
    }
}
```

函数参数并不拥有这个列表，而是通过一个不可变引用来读取它

表示有 1 个 Delta，即 1 单位的势能

这里势能 Δ 的类型为 &Delta，是对一单位势能的不可变引用

消耗势能的函数 tick 要求移动走 *Δ，这并不能做到，因为只有不可变引用的借用权限

- 势能是一种特殊的值，它应当可以通过不可变引用被移动或被消耗
- 在类型系统中，我们需用追踪通过引用发生的“势能的借用”及“势能的归还”

Rust 程序的资源分析：并发抽象

Rust 程序的资源分析：并发抽象

- Rust 强调其并发编程的能力

Rust 程序的资源分析：并发抽象

- Rust 强调其并发编程的能力
  - 可以通过**共享状态**（基于互斥器）、**消息传递**（基于信道）等进行并发

Rust 程序的资源分析：并发抽象

- Rust 强调其并发编程的能力
  - 可以通过**共享状态**（基于互斥器）、**消息传递**（基于信道）等进行并发
  - **一定程度的并发安全**：保证无数据竞争，但**不保证无死锁**

Rust 程序的资源分析：并发抽象

- Rust 强调其并发编程的能力
  - 可以通过**共享状态**（基于互斥器）、**消息传递**（基于信道）等进行并发
  - **一定程度的并发安全**：保证无数据竞争，但**不保证无死锁**

- **问题：死锁显然会破坏资源分析结果的意义（特别是时间复杂度分析）**

Rust 程序的资源分析：并发抽象

```
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```

线程 A

线程 B

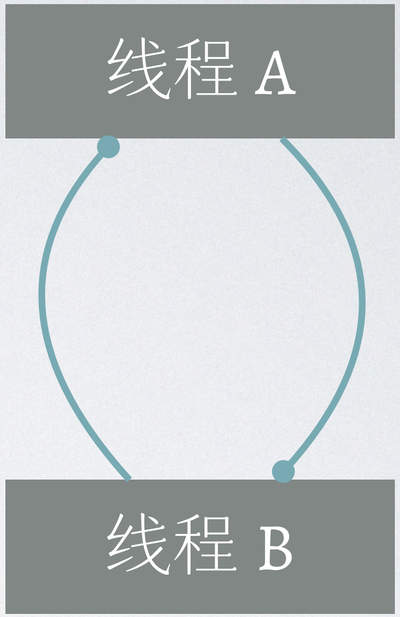
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



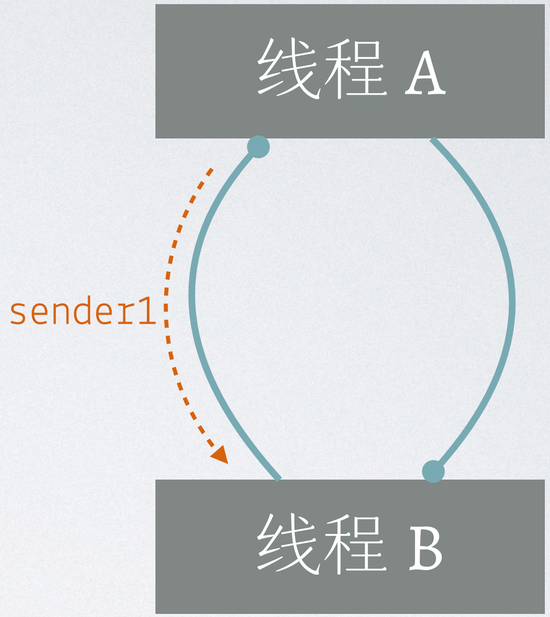
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



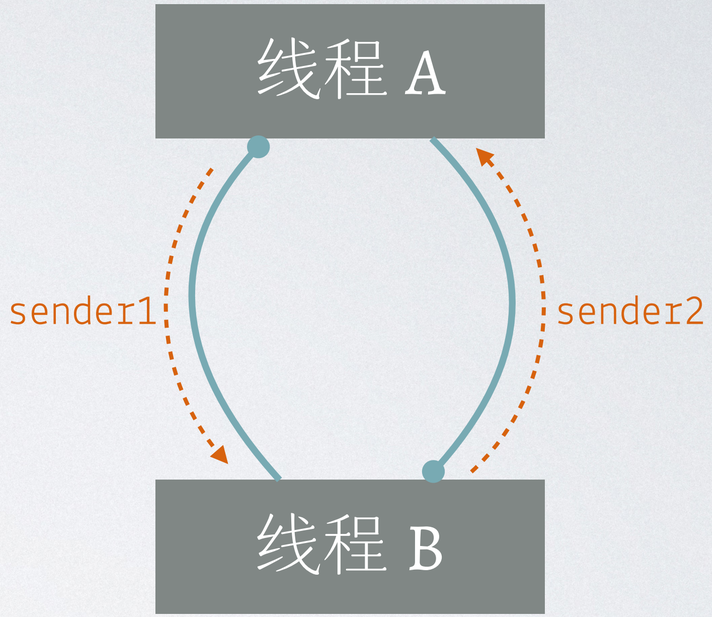
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```rust
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



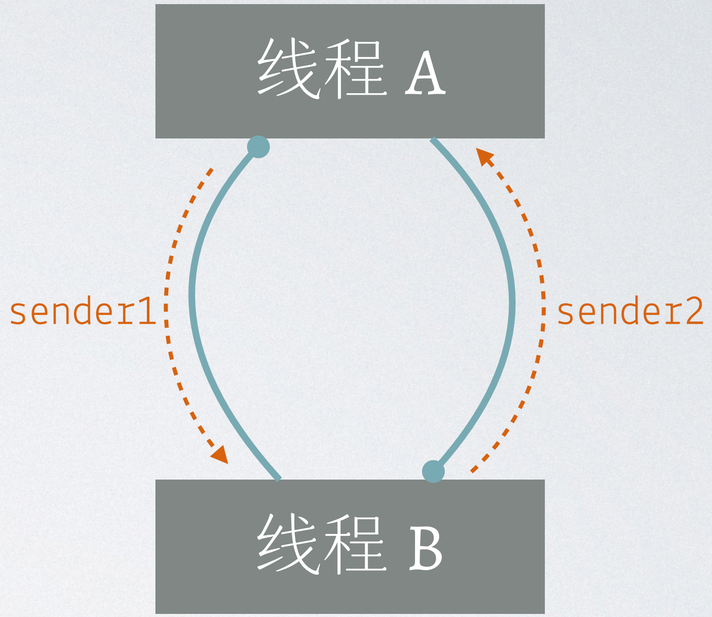
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



- 基于**类型系统**和**资源管理**，设计一套防止死锁的并发机制

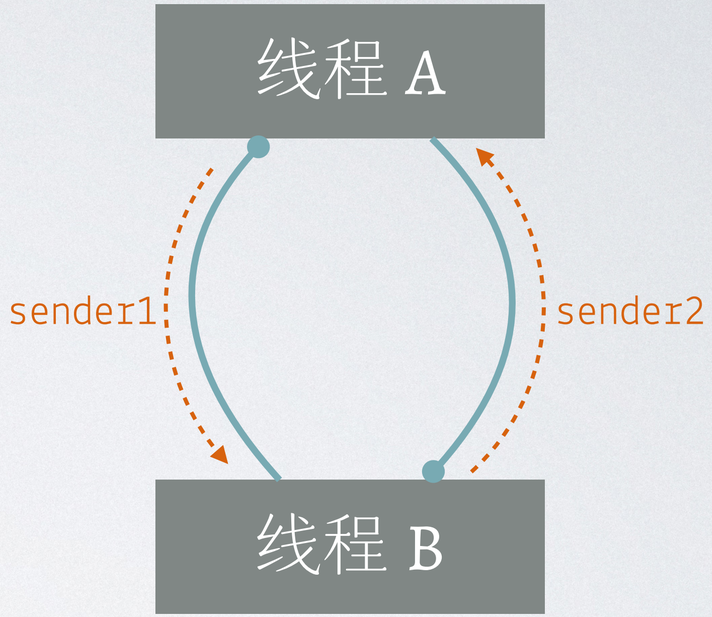
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```rust
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



- 基于**类型系统**和**资源管理**，设计一套防止死锁的并发机制
- 例如，Session Types 要求线程间构成一个**以信道为边的树结构**，从而防止死锁

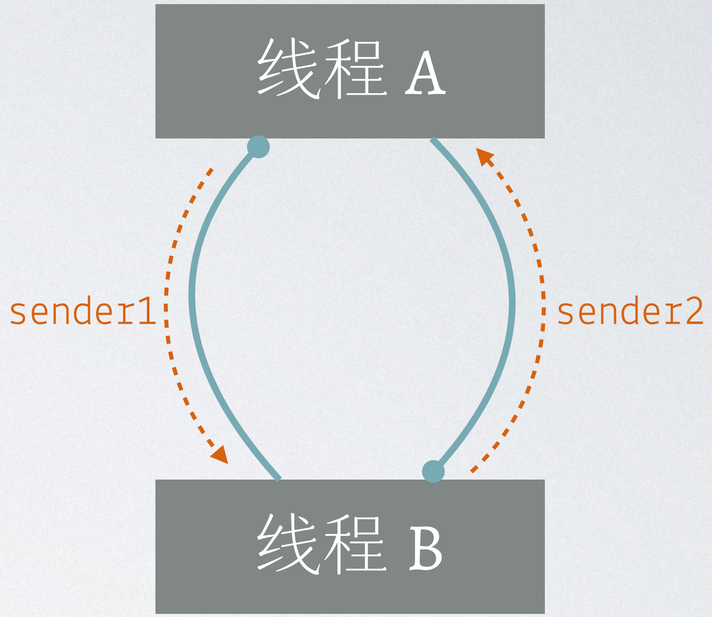
[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：并发抽象

```rust
let (sender1, receiver1) = mpsc::channel::<i32>();
let (sender2, receiver2) = mpsc::channel::<i32>();

thread::spawn(move || {
    let received = receiver2.recv().unwrap();
    sender1.send(received).unwrap();
});

thread::spawn(move || {
    let received = receiver1.recv().unwrap();
    sender2.send(received).unwrap();
});
```



- 基于**类型系统**和**资源管理**，设计一套防止死锁的并发机制
- 例如，Session Types 要求线程间构成一个**以信道为边的树结构**，从而防止死锁
- 已有工作表明，Session Types 和基于势能方法的资源分析可以很好地结合 [1]

[1] A. Das, J. Hoffmann, and F. Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS'18*.

Rust 程序的资源分析：unsafe 机制

Rust 程序的资源分析：unsafe 机制

- Rust 的标准库代码中广泛使用 **`unsafe`**，即绕过类型系统的静态内存安全检查
  - 设计哲学：标准库的开发者需要保证 **`unsafe`** 代码实质上满足内存安全

Rust 程序的资源分析：unsafe 机制

- Rust 的标准库代码中广泛使用 **`unsafe`**，即绕过类型系统的静态内存安全检查
  - 设计哲学：标准库的开发者需要保证 **`unsafe`** 代码实质上满足内存安全
- **问题：基于势能的资源分析依赖于内存安全特性**，比如所有权和移动语义

Rust 程序的资源分析：unsafe 机制

- Rust 的标准库代码中广泛使用 **unsafe**，即绕过类型系统的静态内存安全检查
  - 设计哲学：标准库的开发者需要保证 **unsafe** 代码实质上满足内存安全
- **问题：基于势能的资源分析依赖于内存安全特性**，比如所有权和移动语义
- **可能的解决思路：**
  - 给使用 unsafe 的代码手工标注势能函数（工作量可能很大）
  - 为 unsafe 代码设计资源分析技术（保证分析能力可能很难）

资源安全的系统编程语言

- [x] 算法复杂度符合预期
- [ ] 物理资源消耗满足预期
- [ ] 没有资源相关的安全漏洞

细粒度资源消耗分析

细粒度资源消耗分析

```
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

细粒度资源消耗分析

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

$$O(|src| \cdot n)$$

细粒度资源消耗分析

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

$$O(|src| \cdot n)$$

```rust
fn drop_n_fast(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = Vec::new();
    for i in n..src.len() {
        ret.push(src[i]);
    }
    ret
}
```

$$O(|src|)$$

细粒度资源消耗分析

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

$$O(|src| \cdot n)$$

```rust
fn drop_n_fast(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = Vec::new();
    for i in n..src.len() {
        ret.push(src[i]);
    }
    ret
}
```

$$O(|src|)$$

```rust
fn drop_n_faster(src: &Vec<i32>, n: usize) -> Vec<i32> {
    src.split_at(n).1.into()
}
```

$$O(|src|)$$

细粒度资源消耗分析

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

$$O(|src| \cdot n)$$

```rust
fn drop_n_fast(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = Vec::new();
    for i in n..src.len() {
        ret.push(src[i]);
    }
    ret
}
```

$$O(|src|)$$

```rust
fn drop_n_faster(src: &Vec<i32>, n: usize) -> Vec<i32> {
    src.split_at(n).1.into()
}
```

$$O(|src|)$$

- 虽然 drop_n_fast 和 drop_n_faster 时间复杂度相同，但实测后者能快 3 倍！

细粒度资源消耗分析

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

$$O(|src| \cdot n)$$

```rust
fn drop_n_fast(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = Vec::new();
    for i in n..src.len() {
        ret.push(src[i]);
    }
    ret
}
```

$$O(|src|)$$

```rust
fn drop_n_faster(src: &Vec<i32>, n: usize) -> Vec<i32> {
    src.split_at(n).1.into()
}
```

$$O(|src|)$$

- 虽然 drop_n_fast 和 drop_n_faster 时间复杂度相同，但实测后者能快 3 倍！
- 需要考虑程序**编译后**在具体硬件上运行消耗的物理资源

多种多样的资源度量

多种多样的资源度量

多种多样的资源度量

- 循环次数
- 递归调用次数

多种多样的资源度量

- 循环次数
- 递归调用次数

- 基本代码块数
- MIR 语句数

多种多样的资源度量

- 循环次数
- 递归调用次数

- 基本代码块数
- MIR 语句数

- LLVM 语句数

多种多样的资源度量

- 循环次数
- 递归调用次数

- 基本代码块数
- MIR 语句数

- LLVM 语句数

- 指令数

多种多样的资源度量

- 循环次数
- 递归调用次数

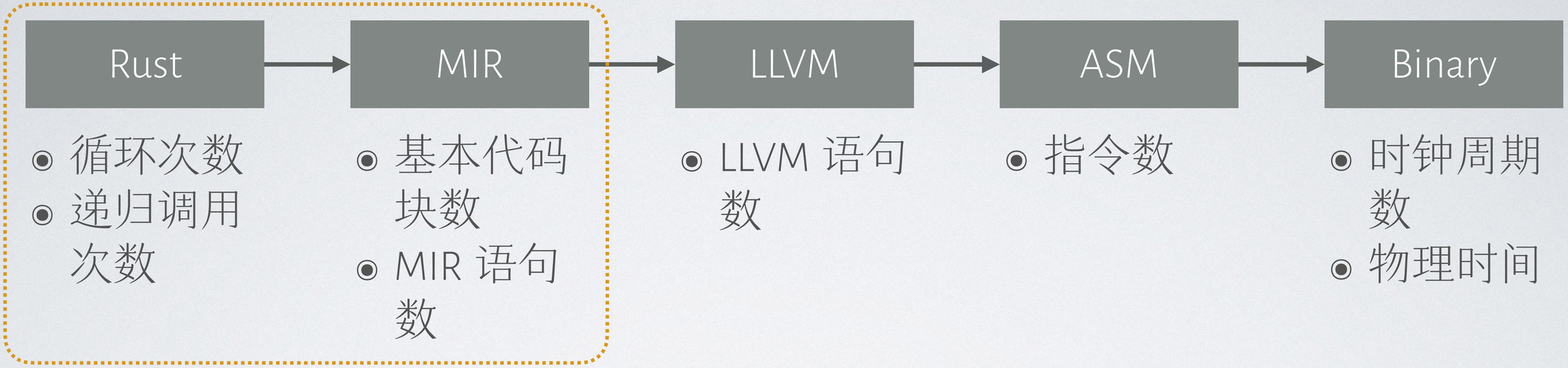
- 基本代码块数
- MIR 语句数

- LLVM 语句数

- 指令数

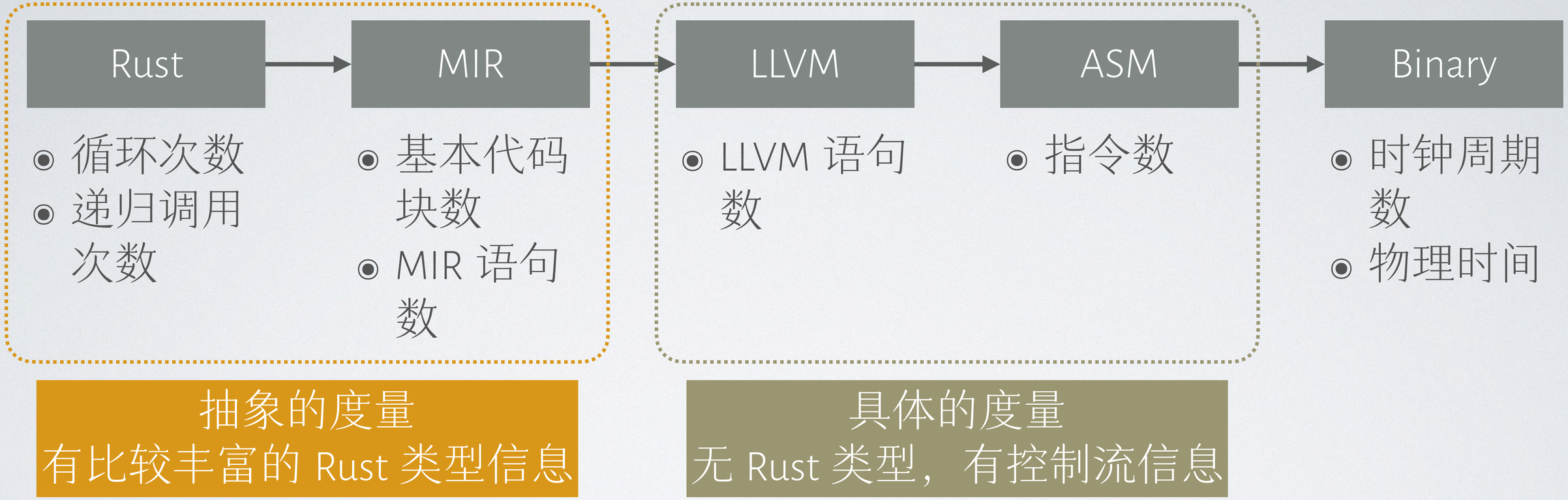
- 时钟周期数
- 物理时间

多种多样的资源度量



抽象的度量
有比较丰富的 Rust 类型信息

多种多样的资源度量


Rust
MIR
LLVM
ASM
Binary
循环次数
递归调用次数
基本代码块数
MIR 语句数
LLVM 语句数
指令数
时钟周期数
物理时间
抽象的度量
有比较丰富的 Rust 类型信息
具体的度量
无 Rust 类型，有控制流信息

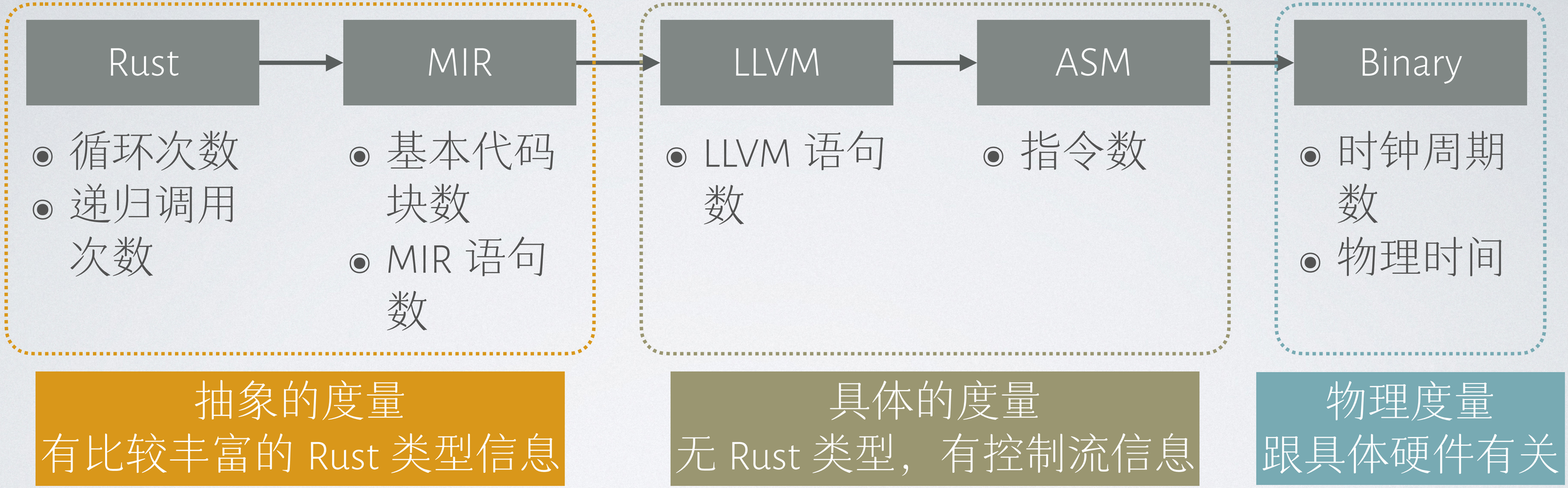
多种多样的资源度量

抽象的度量
有比较丰富的 Rust 类型信息

具体的度量
无 Rust 类型，有控制流信息

物理度量
跟具体硬件有关

多种多样的资源度量



- 随着编译的进行，抽象信息越来越少，资源分析越来越难

多种多样的资源度量

- 随着编译的进行，抽象信息越来越**少**，资源分析越来越**难**
- **问题：如何通过源程序层面的资源分析帮助细粒度的资源分析？**

Worst-Case Execution Time (WCET) [1]

[1] R. Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *Trans. on Embedded Comp. Syst.*, 7, 3.

Worst-Case Execution Time (WCET) [1]

- WCET 最初考虑的是**嵌入式设备**上程序运行最坏情况消耗的**时钟周期数**
  - 这种程序的控制流通常比较简单，比如无递归、循环次数可静态确定

[1] R. Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *Trans. on Embedded Comp. Syst.*, 7, 3.

Worst-Case Execution Time (WCET) [1]

- WCET 最初考虑的是**嵌入式设备**上程序运行最坏情况消耗的**时钟周期数**
  - 这种程序的控制流通常比较简单，比如无递归、循环次数可静态确定

- 商业工具：

AbsInt - aiT

Rapita - RapiTime

[1] R. Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *Trans. on Embedded Comp. Syst.*, 7, 3.

Worst-Case Execution Time (WCET) [1]

- WCET 最初考虑的是**嵌入式设备**上程序运行最坏情况消耗的**时钟周期数**
  - 这种程序的控制流通常比较简单，比如无递归、循环次数可静态确定
- 商业工具：

AbsInt - aiT

Rapita - RapiTime

- 现有工作可以支持对规模较小、控制流较简单的代码分析时钟周期数
  - 也支持用户对比较难分析的部分进行**标注**，比如循环次数上界

[1] R. Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *Trans. on Embedded Comp. Syst.*, 7, 3.

算法复杂度分析 + WCET

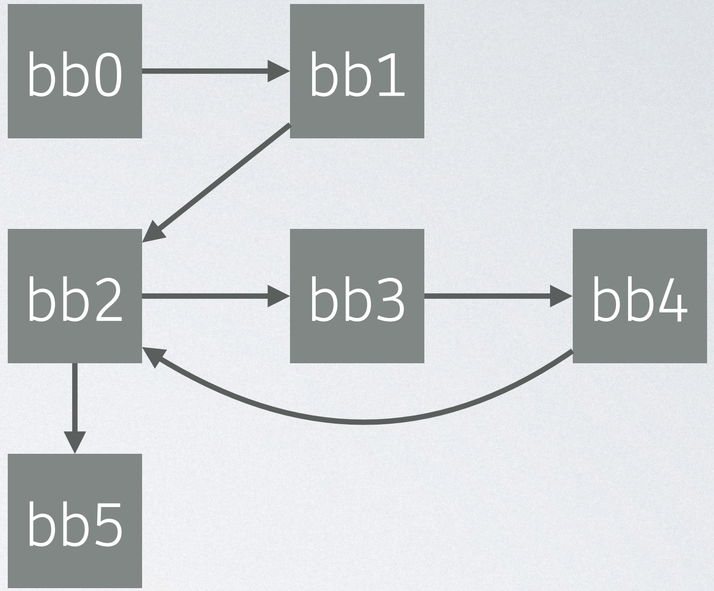
```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

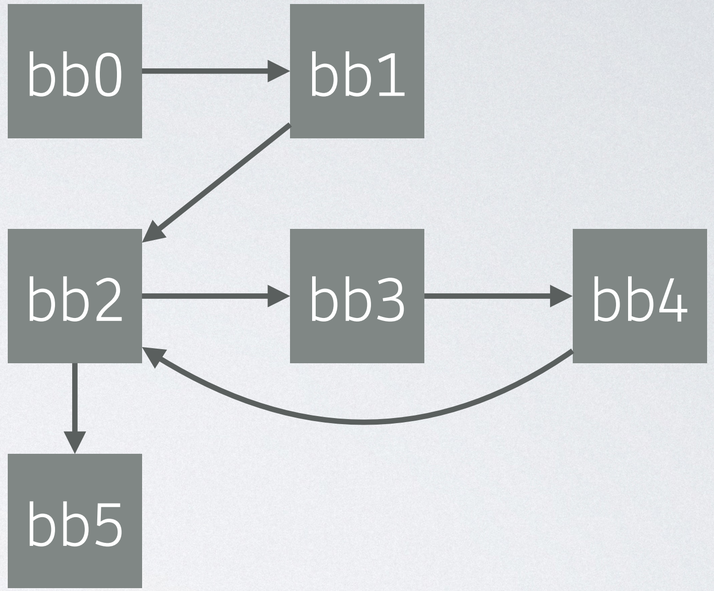


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

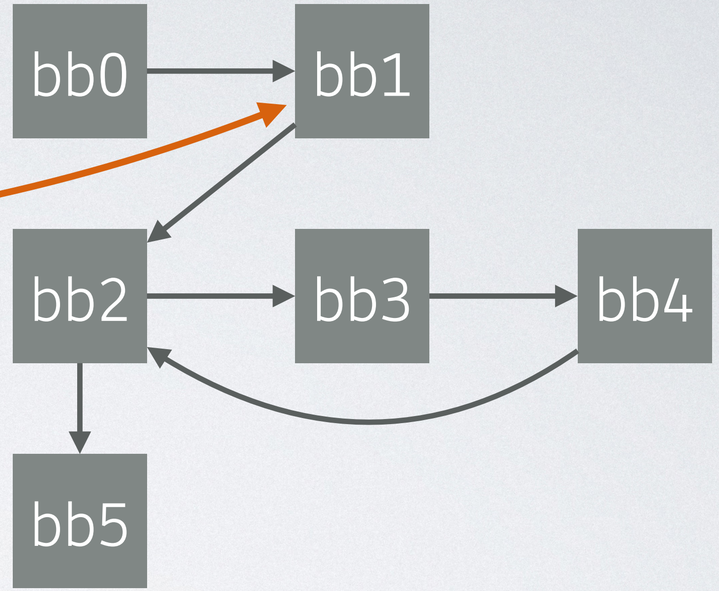


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

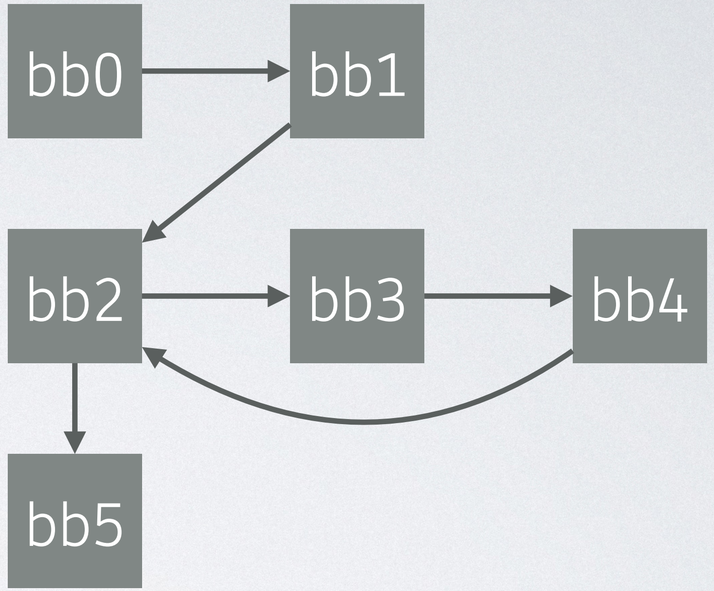


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

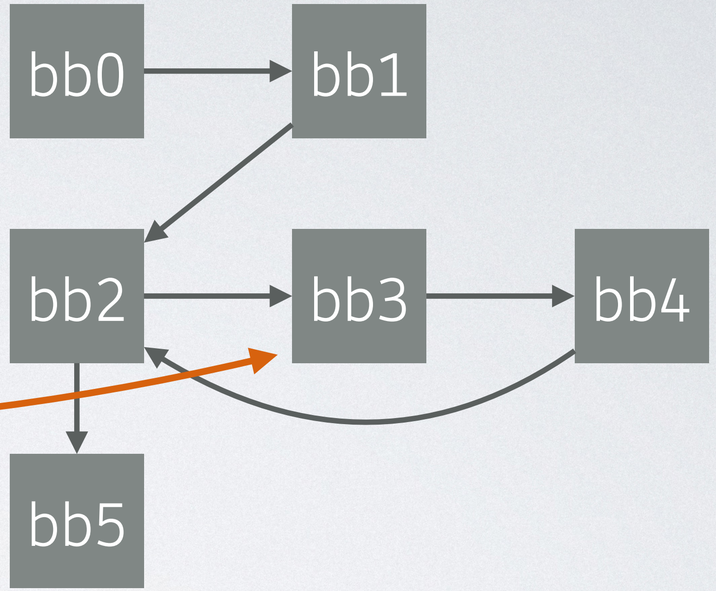


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

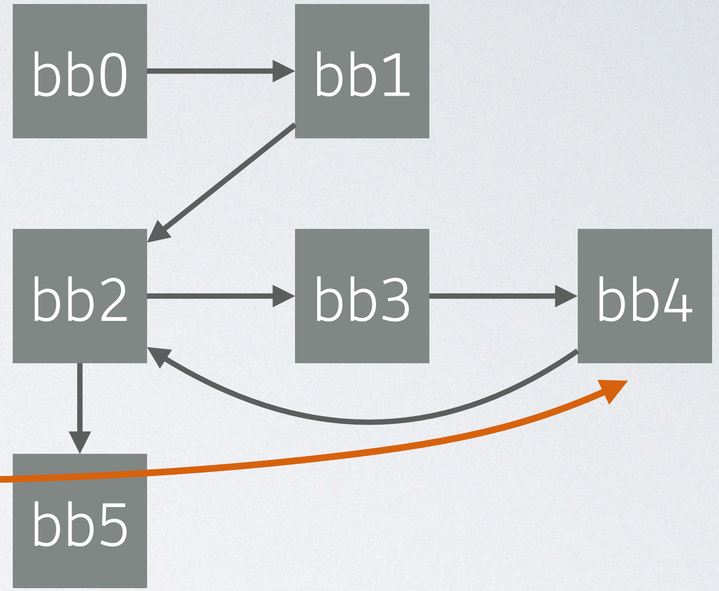


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

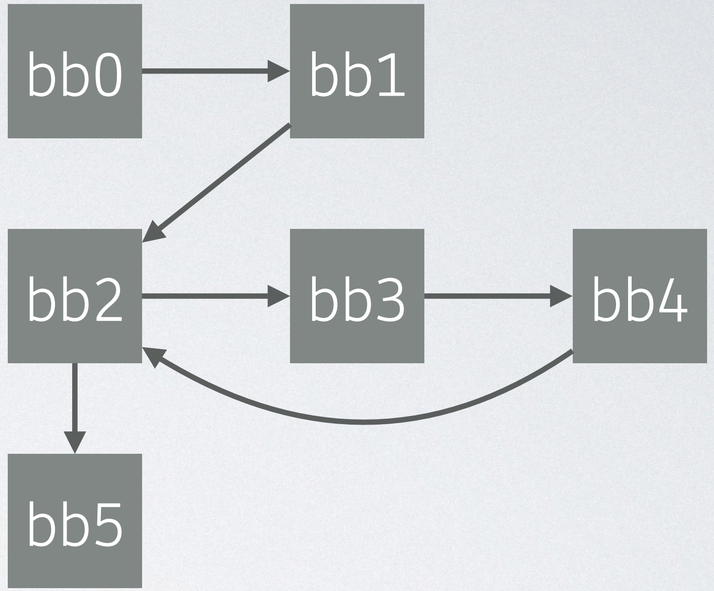


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

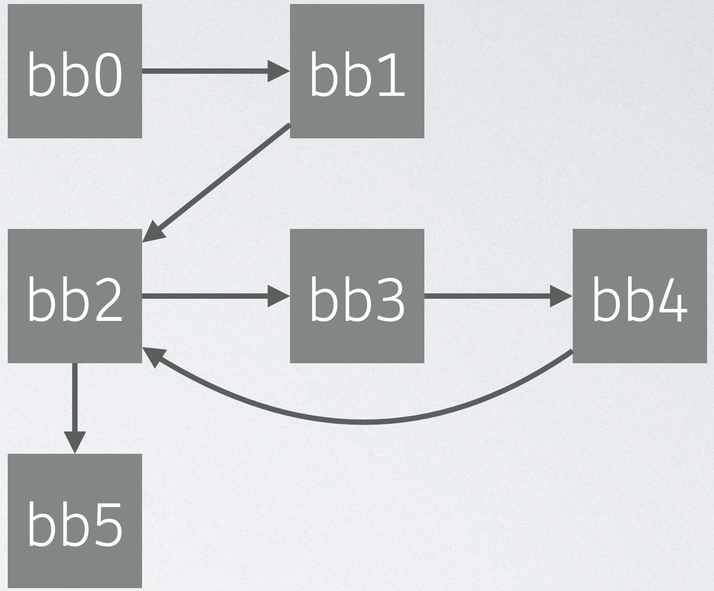


MIR 控制流图

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序



MIR 控制流图

- 每个基本代码块的规模和控制流都比较简单，可以用 WCET 进行分别分析

算法复杂度分析 + WCET

```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

Rust 程序

```rust
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    tick<cost_bb0>();
    let mut i = n;
    tick<cost_bb1>();
    while i != 0 {
        tick<cost_bb2>();
        ret.remove(0);
        tick<cost_bb3>();
        i -= 1;
        tick<cost_bb4>();
    }
    tick<cost_bb2>();
    tick<cost_bb5>();
    ret
}
```

算法复杂度分析 + WCET

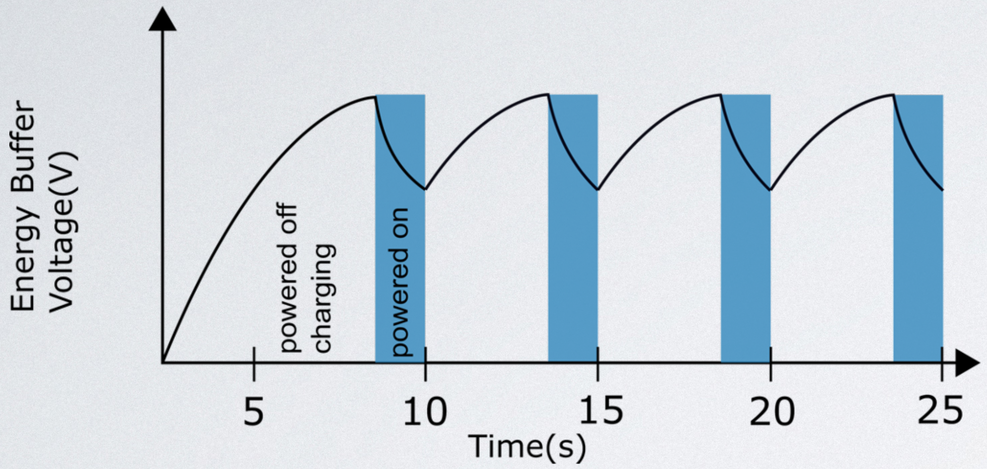
```rust
// 移除src的前n个元素并把结果作为一个新的Vec返回
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    let mut i = n;
    while i != 0 {
        ret.remove(0);
        i -= 1;
    }
    ret
}
```

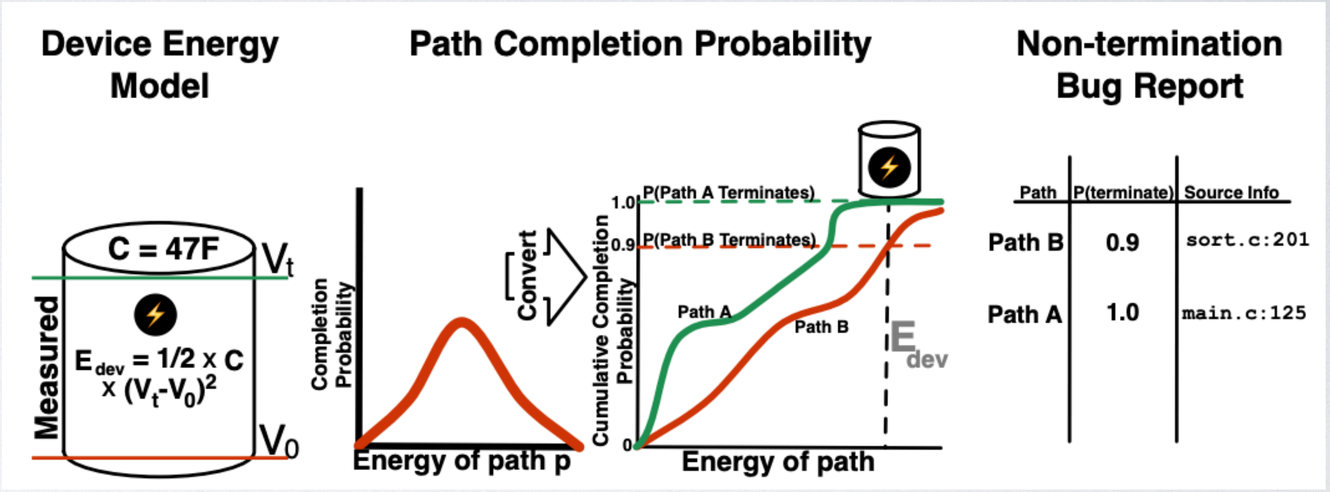
Rust 程序

- 如果编译过程大致**保留了源程序的控制流结构**，可以将 WCET 分析结果接入算法复杂度分析

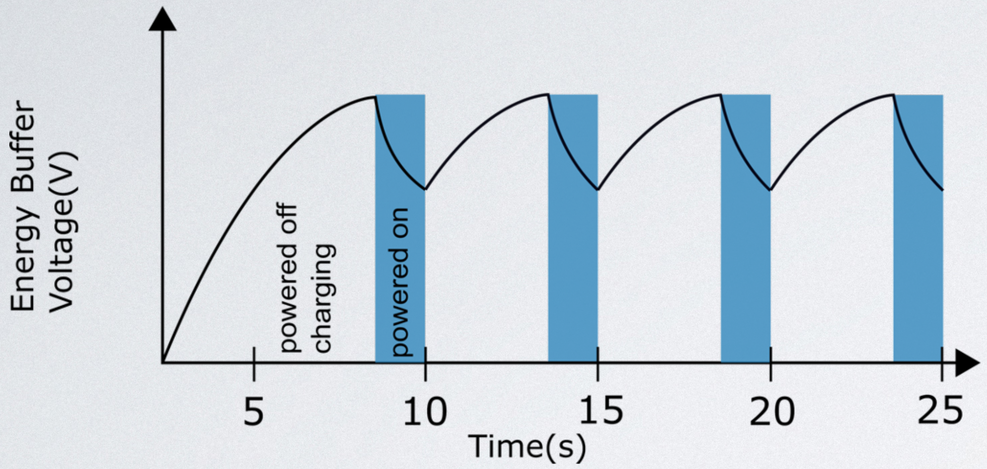
```rust
fn drop_n(src: &Vec<i32>, n: usize) -> Vec<i32> {
    let mut ret = src.clone();
    tick<cost_bb0>();
    let mut i = n;
    tick<cost_bb1>();
    while i != 0 {
        tick<cost_bb2>();
        ret.remove(0);
        tick<cost_bb3>();
        i -= 1;
        tick<cost_bb4>();
    }
    tick<cost_bb2>();
    tick<cost_bb5>();
    ret
}
```

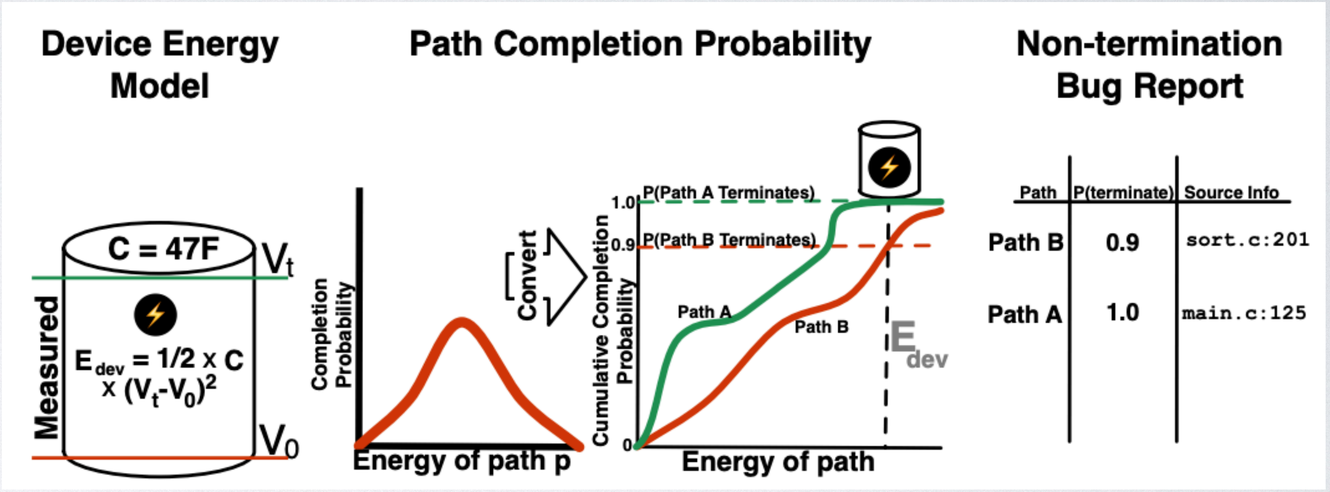
物理度量的不确定性


Energy Buffer
Voltage(V)
powered off
charging
powered on
5
10
15
20
25
Time(s)


Device Energy Model
C = 47F
Vt
Measured
E dev = 1/2 x C x (Vt-V0)2
V0
Path Completion Probability
Completion Probability
Energy of path p
Convert
P(Path A Terminates)
P(Path B Terminates)
Path A
Path B
E dev
Cumulative Completion Probability
1.0
0.9
0
Energy of path
Non-termination Bug Report
Path
P(terminate)
Source Info
Path B
0.9
sort.c:201
Path A
1.0
main.c:125

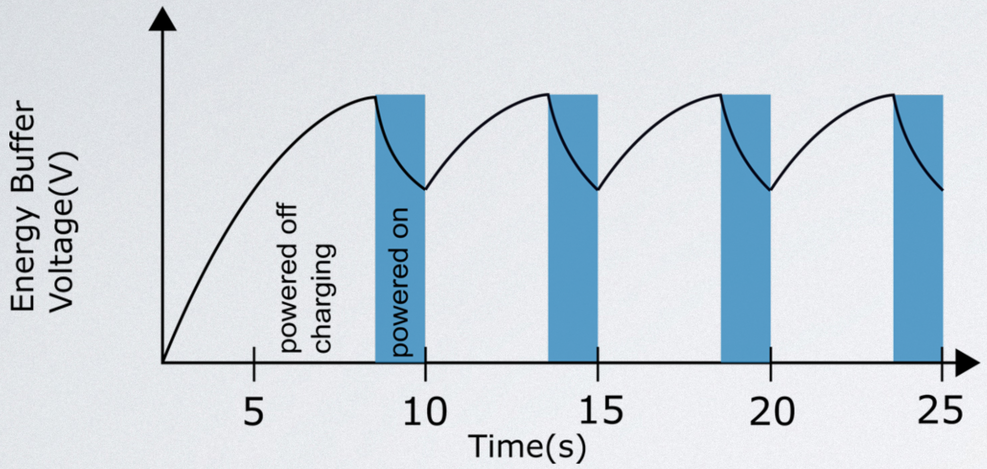
物理度量的不确定性

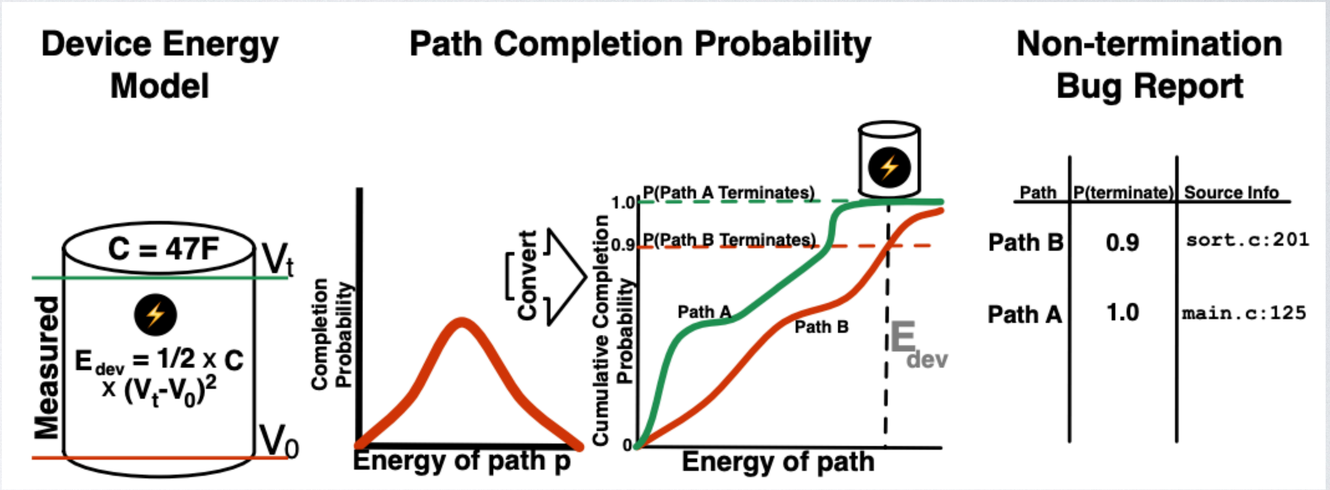




- **能源分析：**同一条指令在相同的设备状态下运行也可能有略微不同的能量消耗

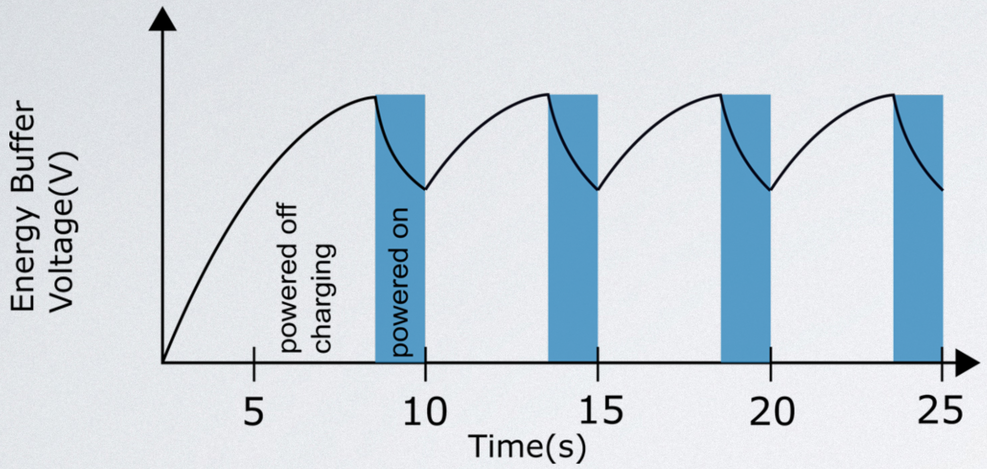
物理度量的不确定性

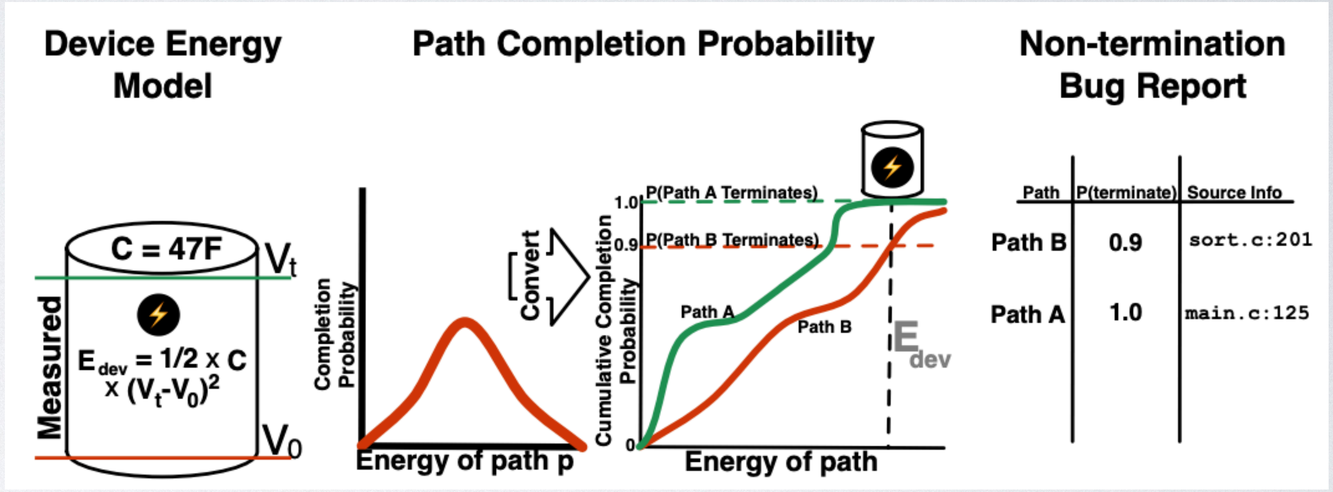




- **能源分析：**同一条指令在相同的设备状态下运行也可能有略微不同的能量消耗
- **时间分析：**程序具体执行的物理时间也会有一些扰动

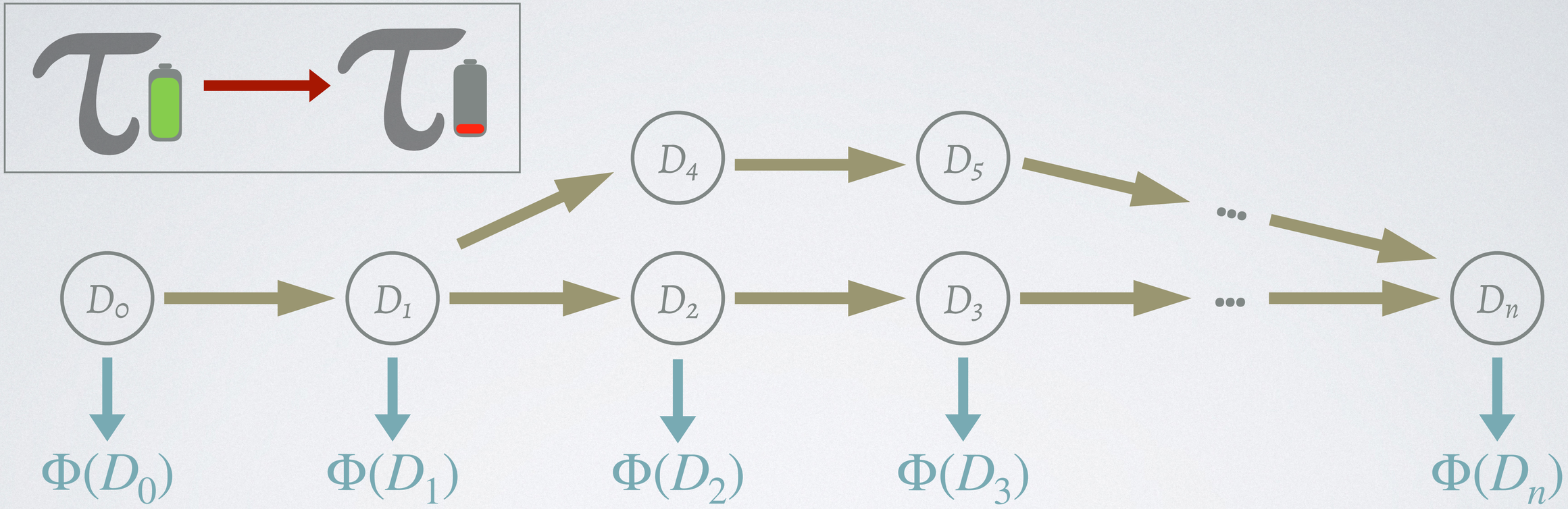
物理度量的不确定性





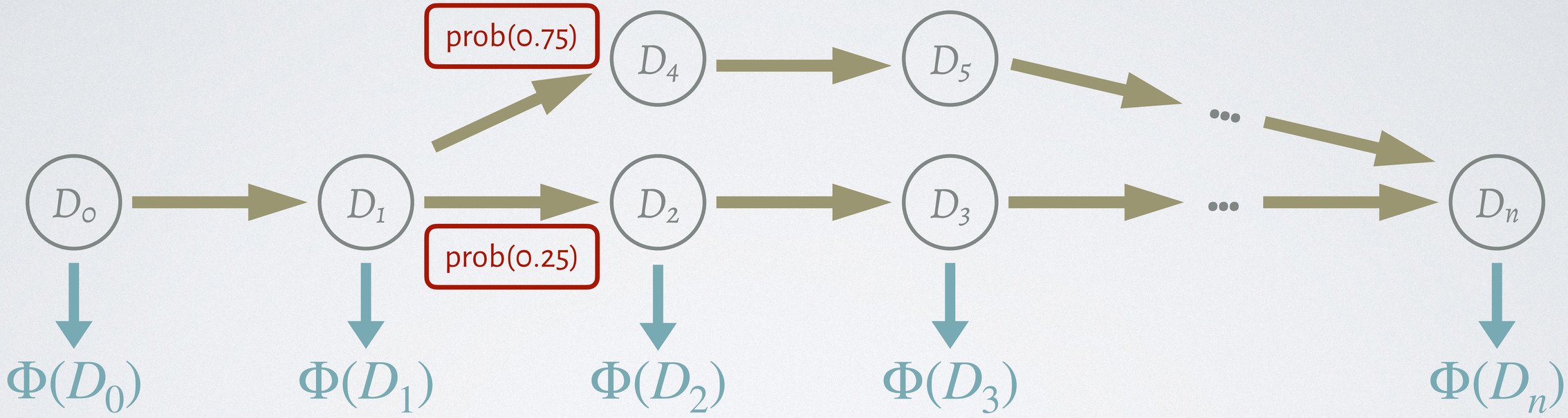
- **能源分析**：同一条指令在相同的设备状态下运行也可能有略微不同的能量消耗
- **时间分析**：程序具体执行的物理时间也会有一些扰动
- 基于**统计或概率分析**，在细粒度的资源分析中加入对不确定性的考量

已完成工作：基于期望势能的分析



已完成工作：基于期望势能的分析

D. Wang, D. M. Kahn, and J. Hoffmann. Raising Expectations: Automating Expected Cost Analysis with Types. In *ICFP'20*.



$$\Phi_E(D_1) \geq 0.75 \cdot (Cost(D_1, D_4) + \Phi_E(D_4)) + 0.25 \cdot (Cost(D_1, D_2) + \Phi_E(D_2))$$

资源安全的系统编程语言

- [x] 算法复杂度符合预期
- [x] 物理资源消耗满足预期
- [ ] 没有资源相关的安全漏洞

资源消耗相关的安全漏洞 [1]

[1] https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

资源消耗相关的安全漏洞[1]

- **算法复杂度攻击**（Algorithmic Complexity Attack, ACA）
  - 构造恶意输入，使得软件消耗过多的资源，从而没有足够的资源响应服务
  - 例如：拒绝服务漏洞

[1] https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

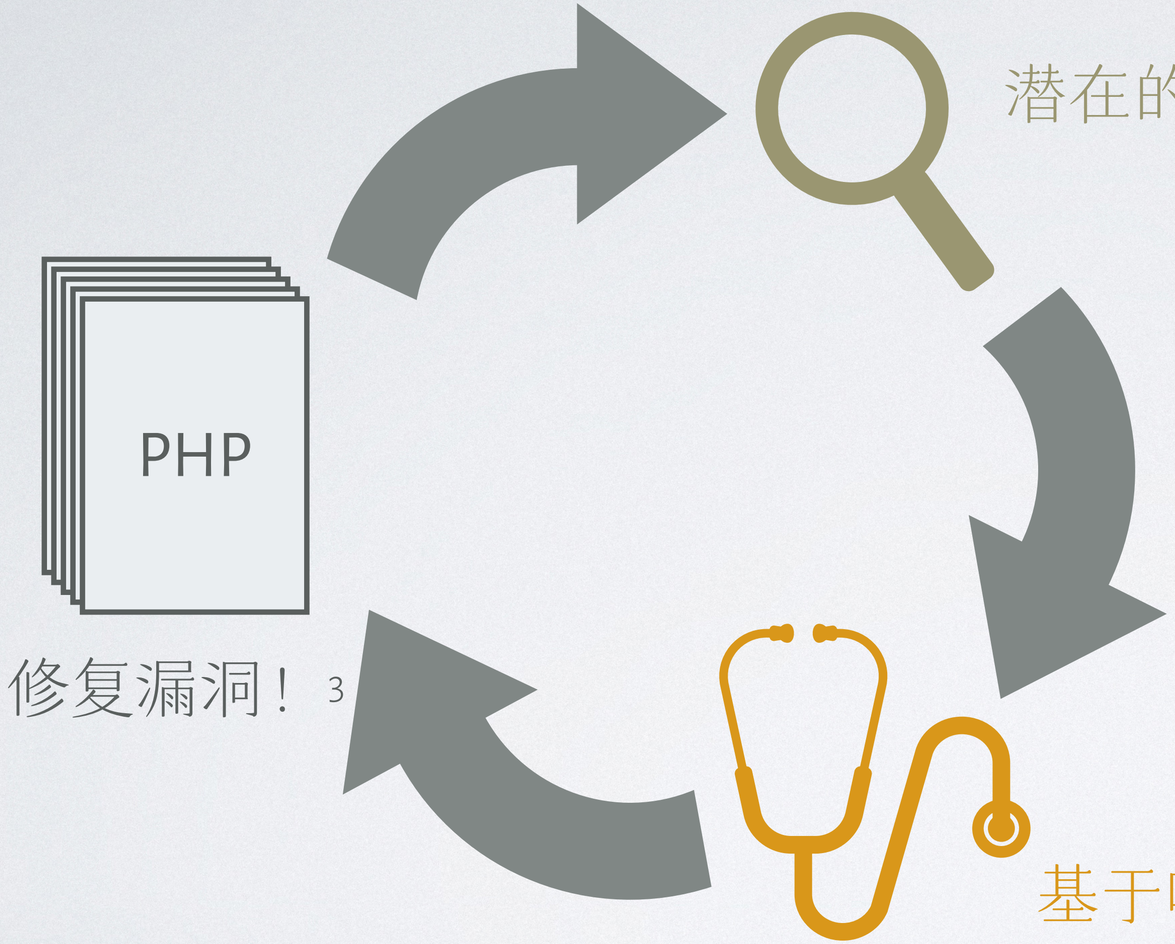
资源消耗相关的安全漏洞[1]

- **算法复杂度攻击**（Algorithmic Complexity Attack, ACA）
  - 构造恶意输入，使得软件消耗过多的资源，从而没有足够的资源响应服务
  - 例如：拒绝服务漏洞

- **侧信道攻击**（Side Channel Attack, SCA）
  - 多次构造输入，收集软件的资源消耗信息，从而猜测出软件中的隐私
  - 例如：破解加密算法密钥的时间攻击漏洞

[1] https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

算法复杂度攻击



潜在的拒绝服务漏洞 [1]

基于哈希冲突的具体攻击实例 [2]

修复漏洞！ [3]

最坏情况测试用例可以帮助开发者发现并修复算法复杂度攻击漏洞

[1] CVE - CVE-2011-4885: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885
[2] PHP 5.3.8 - Hashtables Denial of Service: https://www.exploit-db.com/exploits/18296/
[3] PHP: PHP 5 ChangeLog: http://www.php.net/ChangeLog-5.php#5.3.9

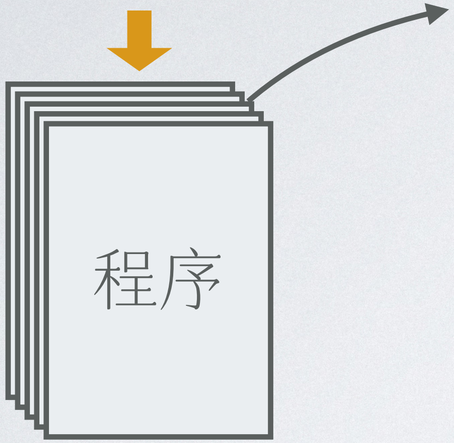
基于符号执行的最坏情况测试生成

基于符号执行的最坏情况测试生成

输入的规约
（例如列表长度）

基于符号执行的最坏情况测试生成

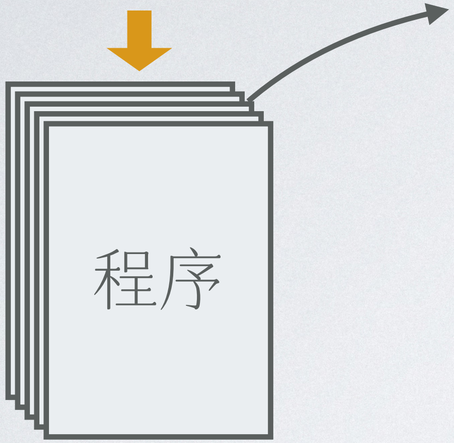
输入的规约
（例如列表长度）



可能的执行路径 1

基于符号执行的最坏情况测试生成

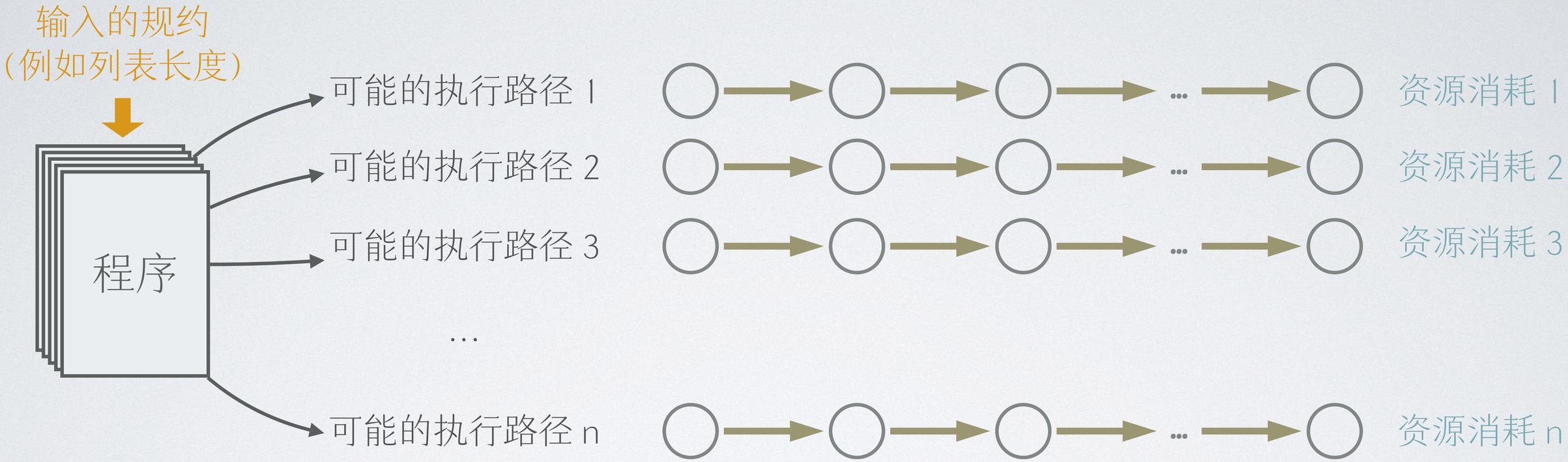
输入的规约
（例如列表长度）



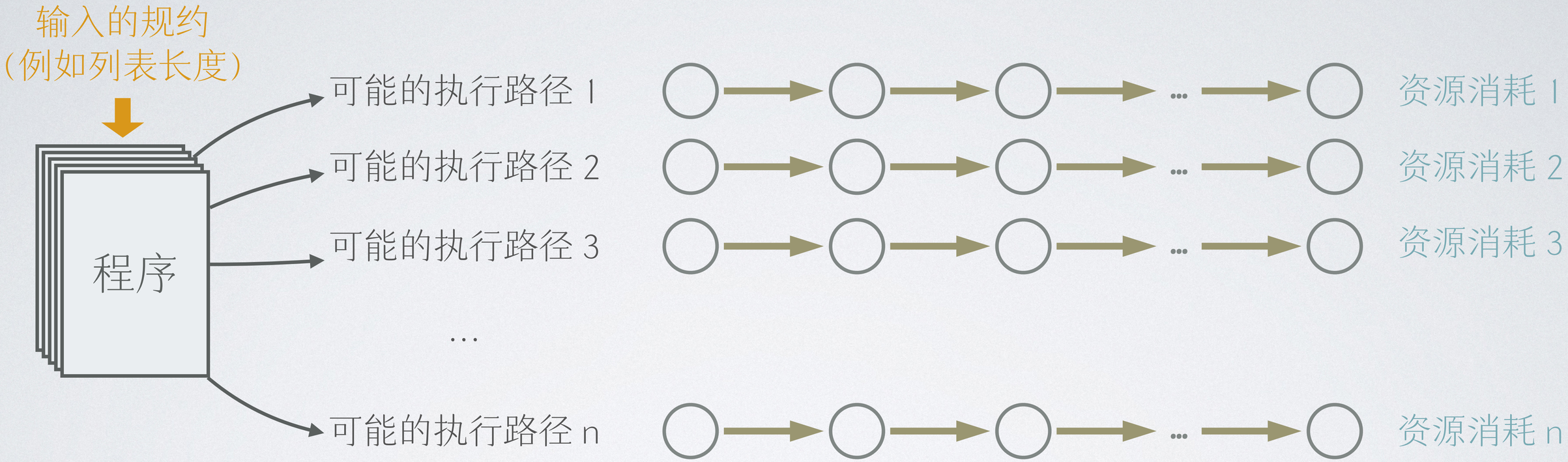
可能的执行路径 1

资源消耗 1

基于符号执行的最坏情况测试生成

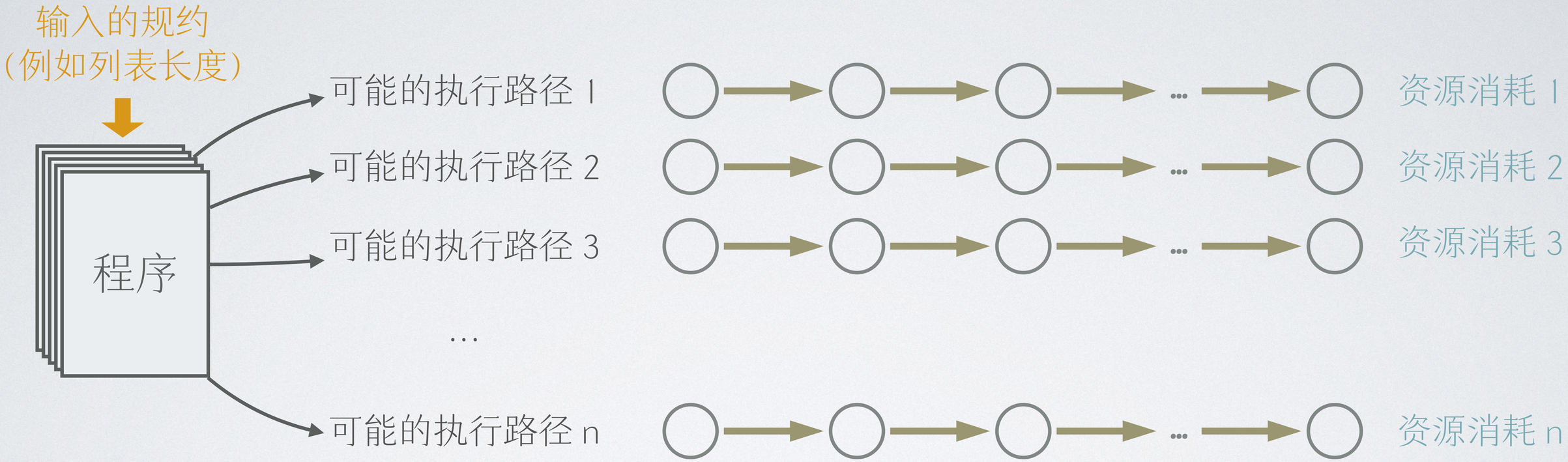


基于符号执行的最坏情况测试生成



- 给定一个输入的规约，搜索一条具有最大资源消耗的程序执行路径

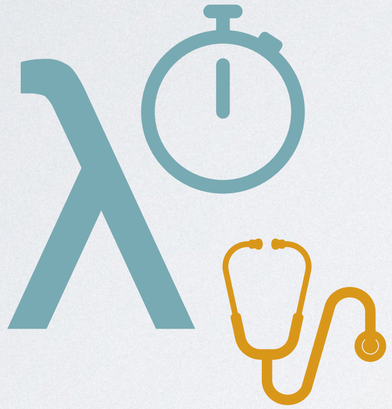
基于符号执行的最坏情况测试生成



- 给定一个输入的规约，搜索一条具有最大资源消耗的程序执行路径
- 是否可以利用静态资源分析的结果来指导符号执行的搜索过程呢？

已完成工作：资源制导最坏情况输入生成

D. Wang and J. Hoffmann. Type-Guided Worst-Case Input Generation. In *POPL'19*.



已完成工作：资源制导最坏情况输入生成

D. Wang and J. Hoffmann. Type-Guided Worst-Case Input Generation. In *POPL'19*.

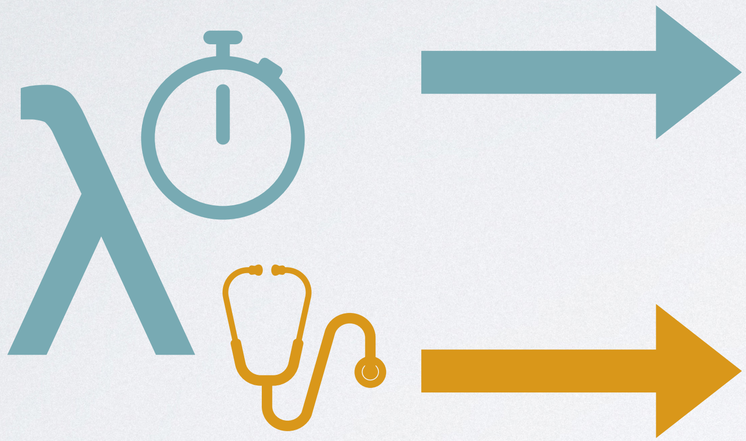
基于势能的自动资源分析

指导

基于符号执行的最坏情况输入生成

已完成工作：资源制导最坏情况输入生成

D. Wang and J. Hoffmann. Type-Guided Worst-Case Input Generation. In *POPL'19*.



基于势能的自动资源分析

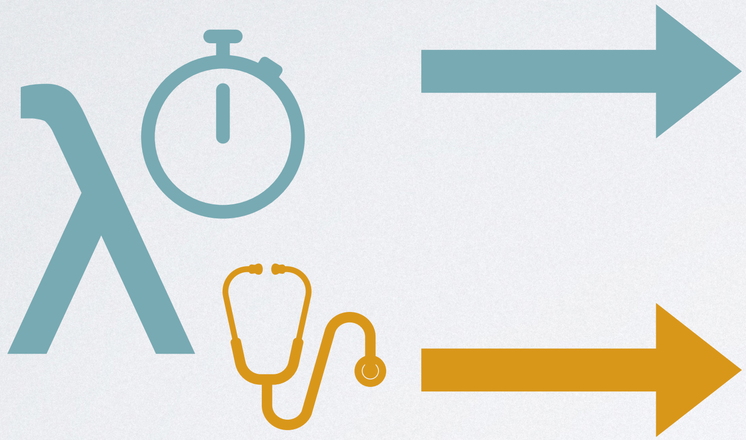
指导

基于符号执行的最坏情况输入生成

首个**有理论正确性**保证的，基于**静态资源分析**的
最坏情况输入生成算法

已完成工作：资源制导最坏情况输入生成

D. Wang and J. Hoffmann. Type-Guided Worst-Case Input Generation. In *POPL'19*.



基于势能的自动资源分析

指导

基于符号执行的最坏情况输入生成

如果一条程序执行路径**没有**任何**势能浪费**，那么它一定具有最坏的资源消耗

首个**有理论正确性**保证的，基于**静态资源分析**的**最坏情况输入**生成算法

侧信道攻击

```
func compare(guess, secret) {
  i := N; cmp := 0;
  while i > 0 {
    tick(2);
    while i > 0 {
      if prob(0.5) break;
      tick(5);
      if cmp > 0 || (cmp == 0 && guess[i] > secret[i]) {
        cmp := 1;
      } else {
        tick(5);
        if cmp < 0 || (cmp == 0 && guess[i] < secret[i]) {
          cmp := -1;
        }
      }
      tick(1);
      i := i - 1;
    }
  }
  return cmp;
}
```

compare 建模了一个密码检查函数，guess 是用户输入，secret 是软件中存储的密钥（都是 N 位的）

侧信道攻击

```
func compare(guess, secret) {
  i := N; cmp := 0;
  while i > 0 {
    tick(2);
    while i > 0 {
      if prob(0.5) break;
      tick(5);
      if cmp > 0 || (cmp == 0 && guess[i] > secret[i]) {
        cmp := 1;
      } else {
        tick(5);
        if cmp < 0 || (cmp == 0 && guess[i] < secret[i]) {
          cmp := -1;
        }
      }
      tick(1);
      i := i - 1;
    }
  }
  return cmp;
}
```

prob 表达式引入了**随机行为**：该函数通过给运行时间加入**随机扰动**来（企图）防止侧信道攻击

compare 建模了一个密码检查函数，guess 是用户输入，secret 是软件中存储的密钥（都是 N 位的）

侧信道攻击

```
func compare(guess, secret) {
  i := N; cmp := 0;
  while i > 0 {
    tick(2);
    while i > 0 {
      if prob(0.5) break;
      tick(5);
      if cmp > 0 || (cmp == 0 && guess[i] > secret[i]) {
        cmp := 1;
      } else {
        tick(5);
        if cmp < 0 || (cmp == 0 && guess[i] < secret[i]) {
          cmp := -1;
        }
      }
      tick(1);
      i := i - 1;
    }
  }
  return cmp;
}
```

prob 表达式引入了**随机行为**：该函数通过给运行时间加入**随机扰动**来（企图）防止侧信道攻击

compare 建模了一个密码检查函数，guess 是用户输入，secret 是软件中存储的密钥（都是 N 位的）

攻击方法：是否可以通过调用 compare 多次并**测量运行时间**，从而猜出 secret 的值？

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

$$13N - 5K \leq \mathbb{E}[T_0] \leq 13N - 3K \qquad 13N \leq \mathbb{E}[T_1] \leq 15N$$

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

$13N - 5K \le \mathbb{E}[T_0] \le 13N - 3K$ $\qquad$ $13N \le \mathbb{E}[T_1] \le 15N$

如果运行时间小于 $13N - 1.5K$，那么下一位很有可能是 0

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

$$13N - 5K \le \mathbb{E}[T_0] \le 13N - 3K \qquad 13N \le \mathbb{E}[T_1] \le 15N$$

如果运行时间小于 $13N - 1.5K$，那么下一位很有可能是 0

$$\mathbb{E}[(T_0 - \mathbb{E}[T_0])^2] \le 8N - 36K^2 + 52NK + 24K \qquad \mathbb{E}[(T_1 - \mathbb{E}[T_1])^2] \le 26N^2 + 42N$$

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

$$13N - 5K \le \mathbb{E}[T_0] \le 13N - 3K \qquad 13N \le \mathbb{E}[T_1] \le 15N$$

如果运行时间小于 $13N - 1.5K$，那么下一位很有可能是 0

$$\mathbb{E}[(T_0 - \mathbb{E}[T_0])^2] \le 8N - 36K^2 + 52NK + 24K \qquad \mathbb{E}[(T_1 - \mathbb{E}[T_1])^2] \le 26N^2 + 42N$$

通过中心矩（如方差）和集中不等式（如马尔科夫不等式）可以分析

$\mathbb{P}[T_1 \le 13N - 1.5K]$ 和 $\mathbb{P}[T_0 \ge 13N - 1.5K]$ 的上界

已完成工作：概率程序资源消耗高阶矩分析

D. Wang, J. Hoffmann, and T. Reps. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *PLDI'21*.

$T_1 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 1 情况下的运行时间

$T_0 \stackrel{\text{def}}{=}$ 假设攻击者已经获得了前 K 位，下一位是 0 情况下的运行时间

$$13N - 5K \le \mathbb{E}[T_0] \le 13N - 3K \qquad 13N \le \mathbb{E}[T_1] \le 15N$$

如果运行时间小于 $13N - 1.5K$，那么下一位很有可能是 0

$$\mathbb{E}[(T_0 - \mathbb{E}[T_0])^2] \le 8N - 36K^2 + 52NK + 24K \qquad \mathbb{E}[(T_1 - \mathbb{E}[T_1])^2] \le 26N^2 + 42N$$

通过中心矩（如方差）和集中不等式（如马尔科夫不等式）可以分析

$\mathbb{P}[T_1 \le 13N - 1.5K]$ 和 $\mathbb{P}[T_0 \ge 13N - 1.5K]$ 的上界

$\mathbb{P}$[攻击者成功] ≥ 0.830561 若 $N = 32$

资源安全的系统编程语言

- ☑ 算法复杂度符合预期
- ☑ 物理资源消耗满足预期
- ☑ 没有资源相关的安全漏洞

资源安全的系统编程语言

- ☑ 算法复杂度符合预期
- ☑ 物理资源消耗满足预期
- ☑ 没有资源相关的安全漏洞

- Rust 的流行说明了在类型系统中追踪**更多的安全性质**对系统编程语言是有益的

资源安全的系统编程语言

- ☑ 算法复杂度符合预期
- ☑ 物理资源消耗满足预期
- ☑ 没有资源相关的安全漏洞

- Rust 的流行说明了在类型系统中追踪**更多的安全性质**对系统编程语言是有益的
- 如何**从资源安全的需求出发**，改进 Rust 或者设计新的系统编程语言？

谢谢

wangdi95@pku.edu.cn