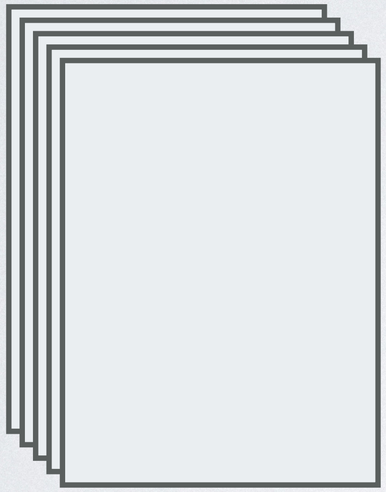
程序的资源分析与验证

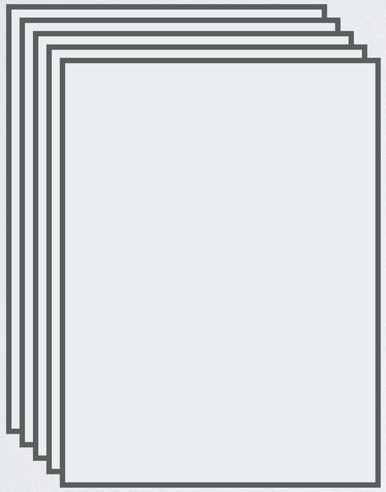
王迪
北京大学
wangdi95@pku.edu.cn
2024 年 11 月 30 日

程序的资源消耗



程序

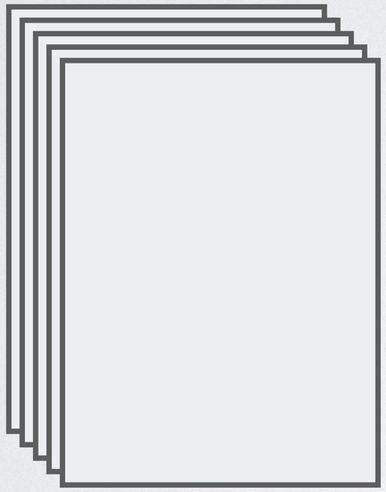
程序的资源消耗



程序

性能

程序的资源消耗

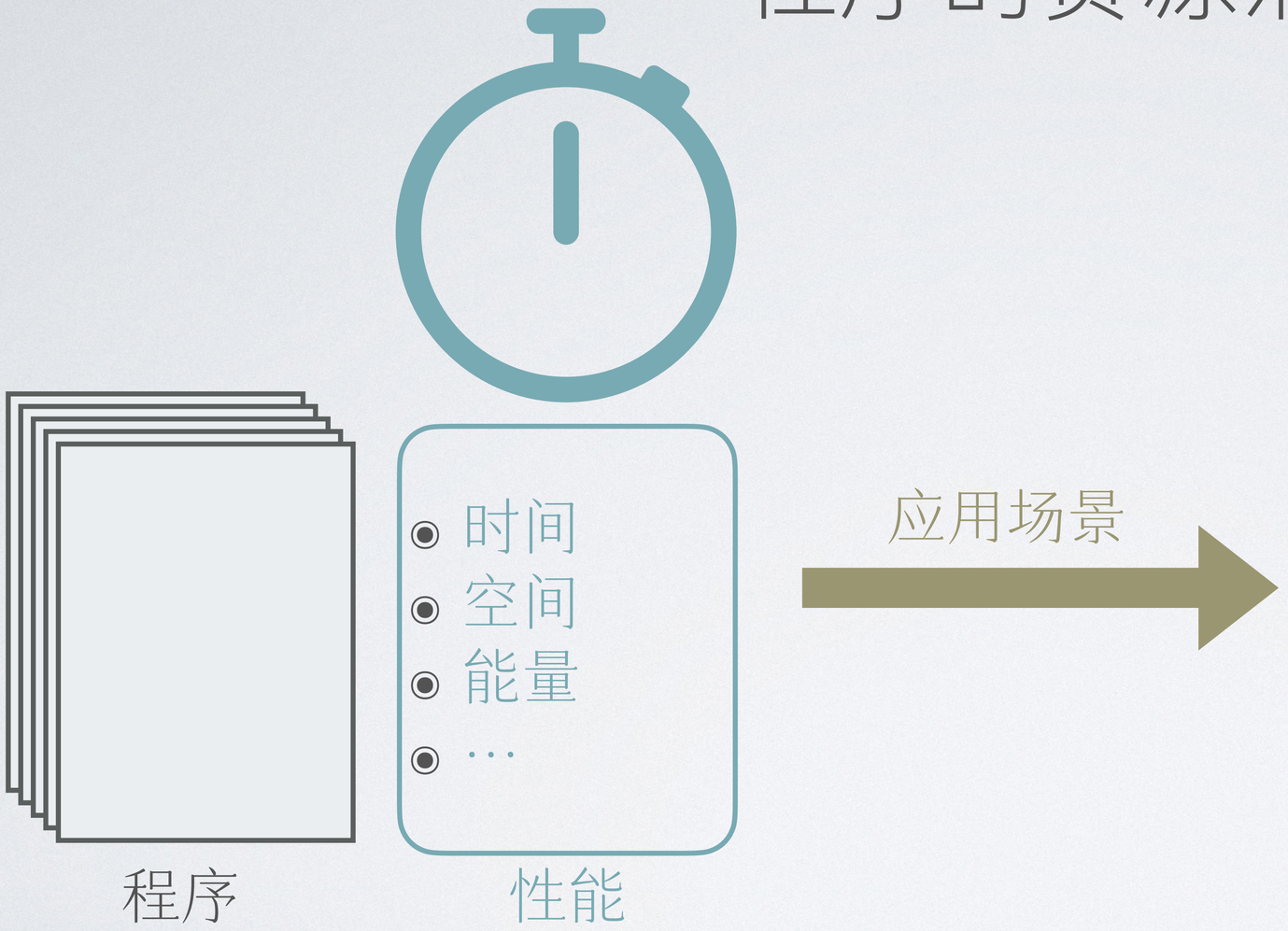


- 时间
- 空间
- 能量
- …

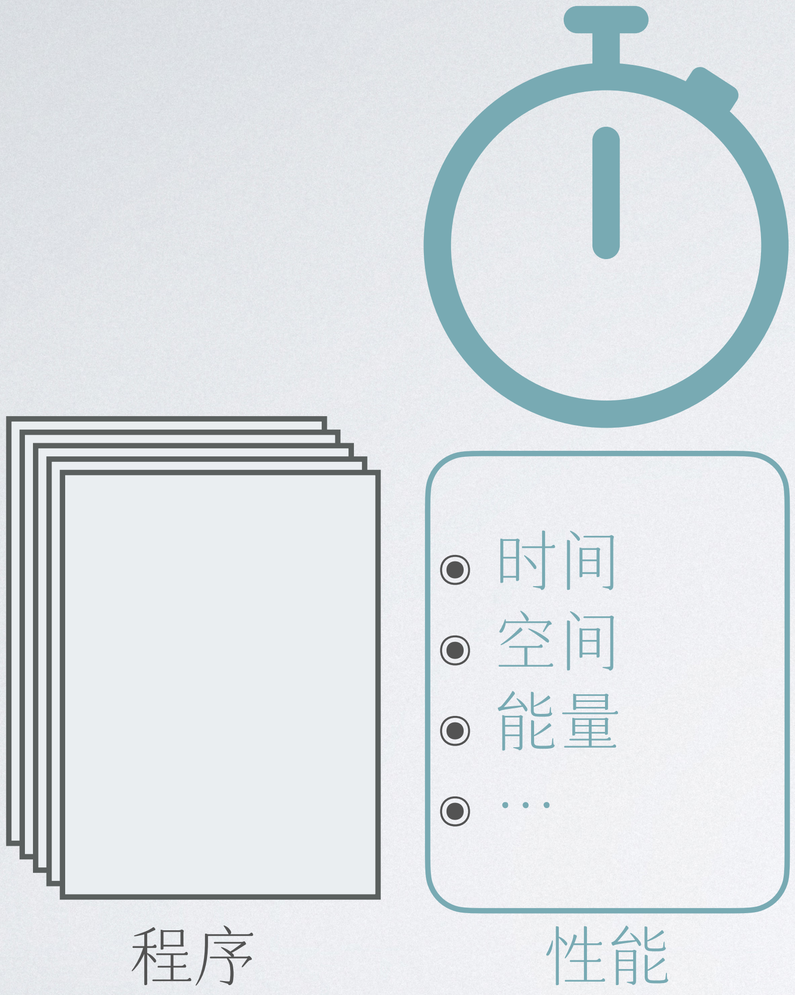
程序

性能

程序的资源消耗



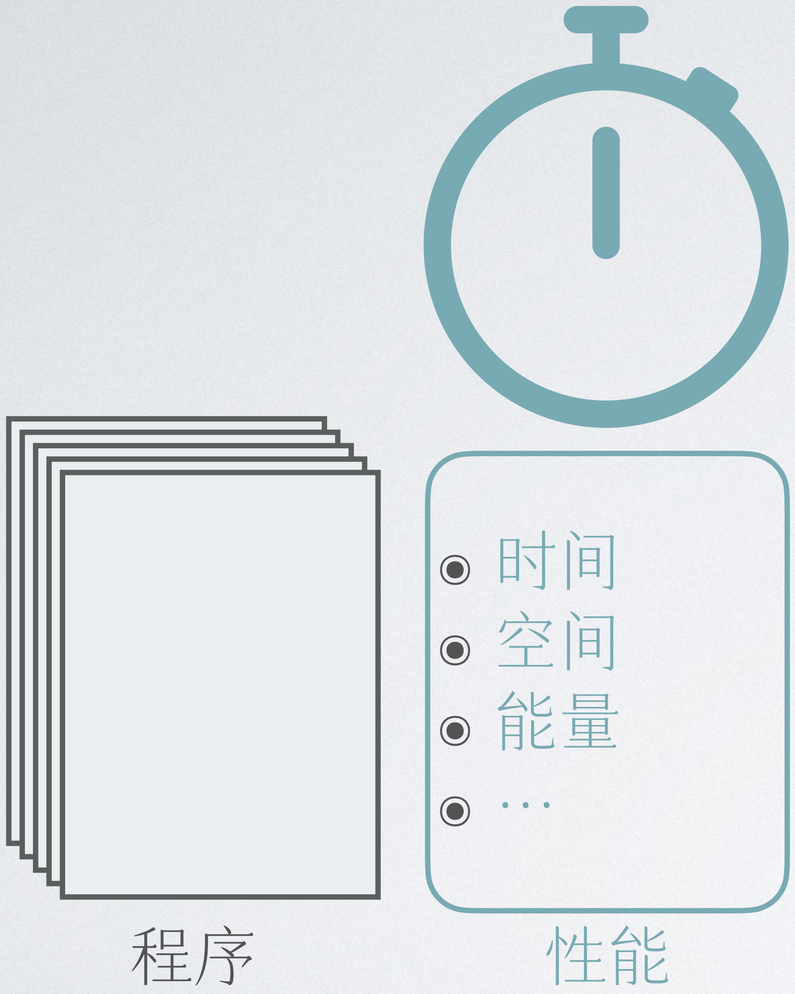
程序的资源消耗



应用场景

- 分析算法的时间复杂度

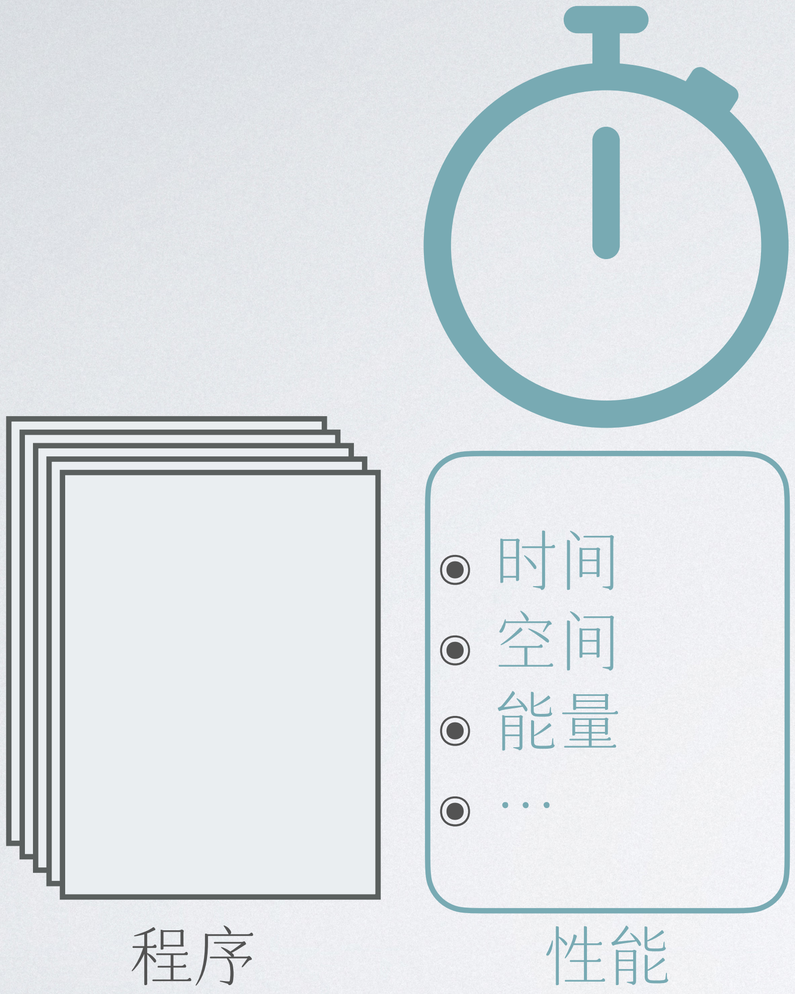
程序的资源消耗



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

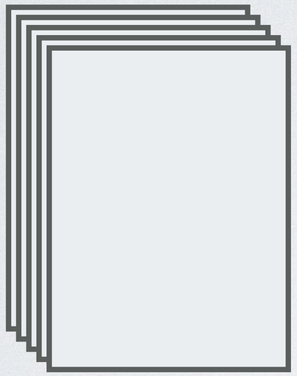
程序的资源消耗



应用场景

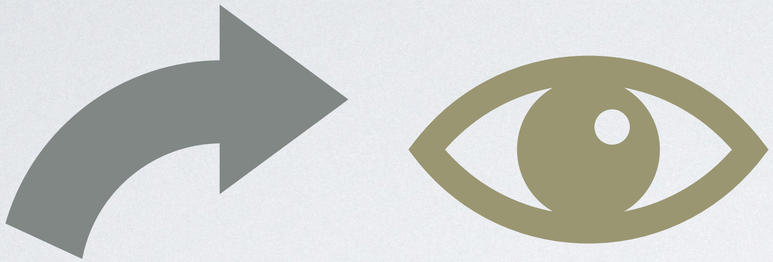
- 分析算法的时间复杂度
- 预测智能合约的燃气消耗
- 发现软件的拒绝服务漏洞

资源分析的时机

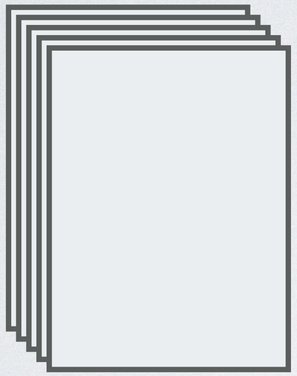


软件开发

资源分析的时机

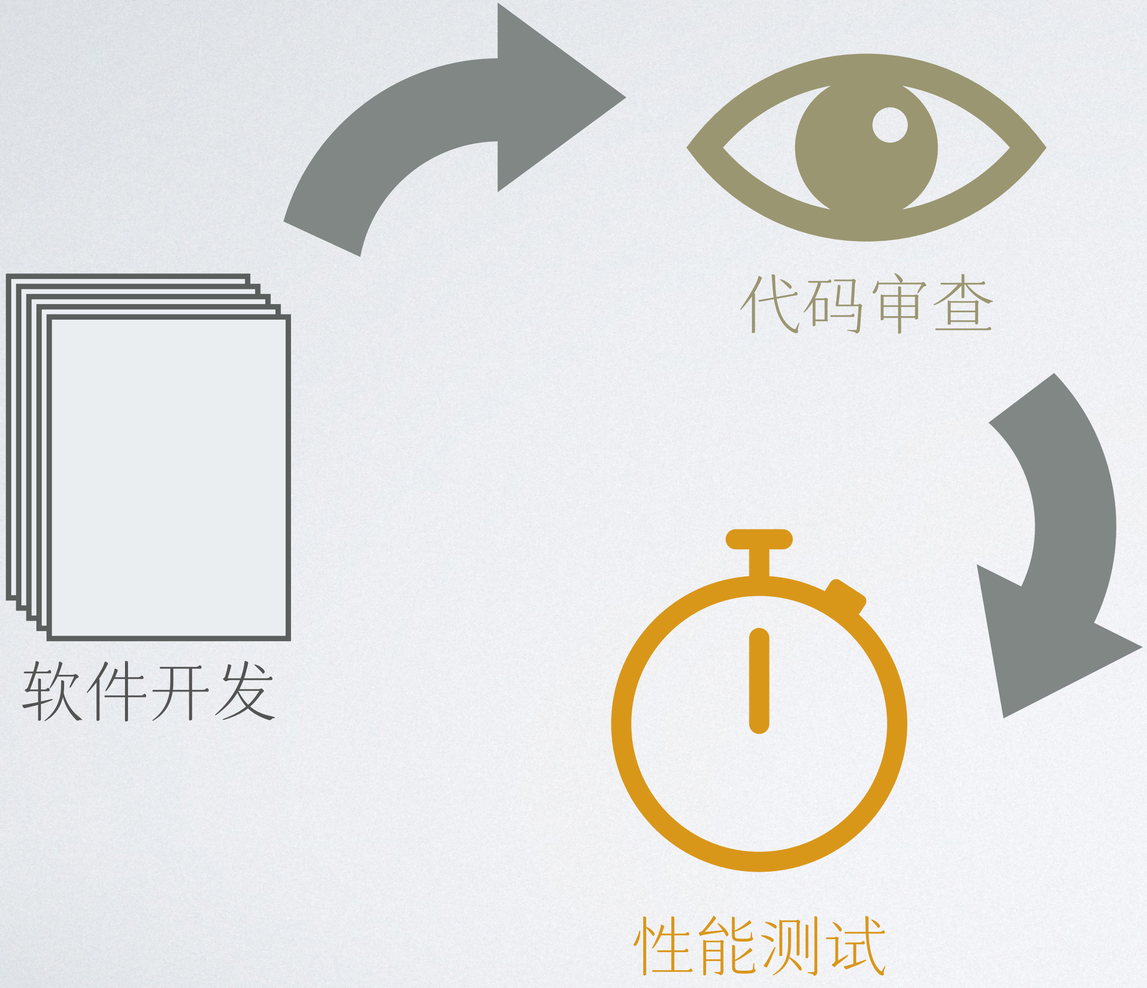


代码审查

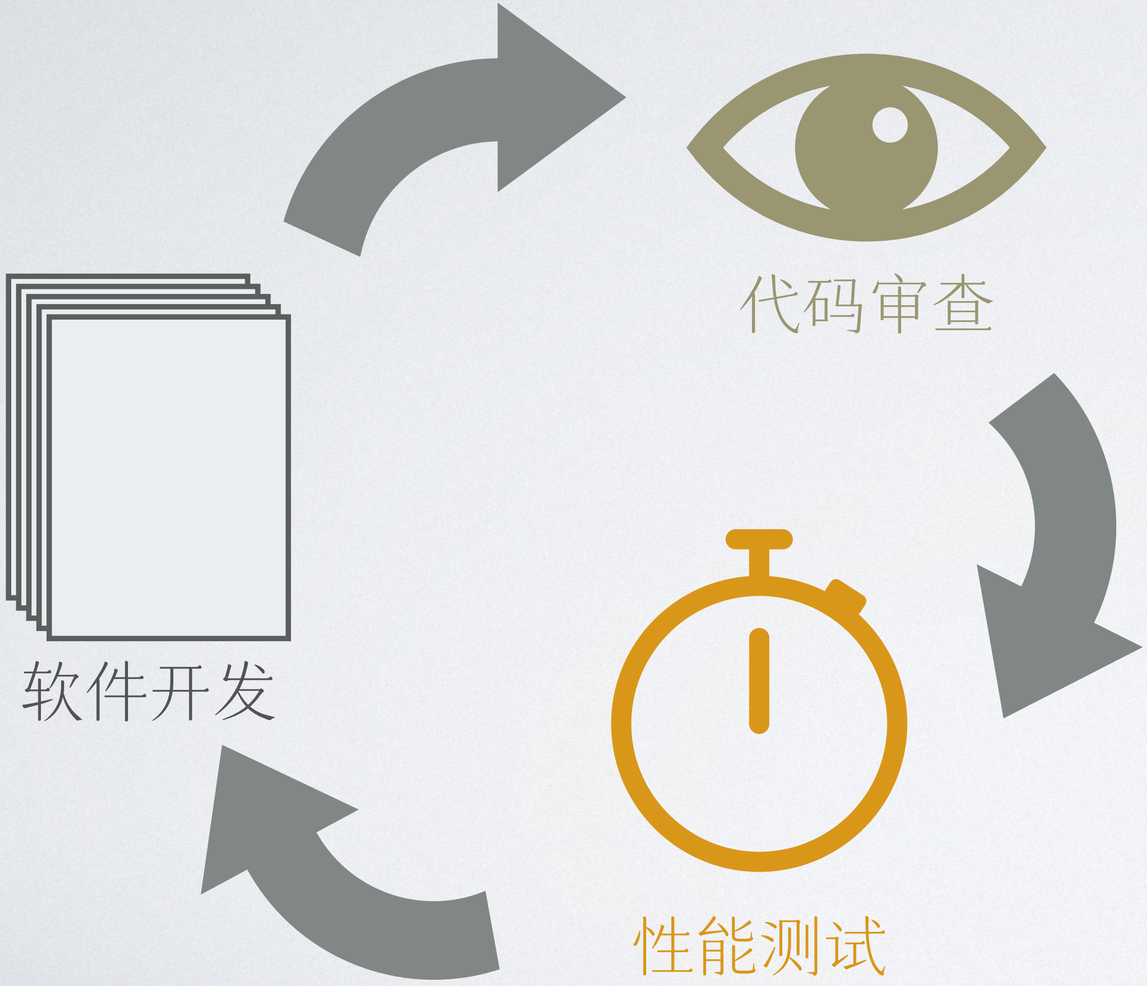


软件开发

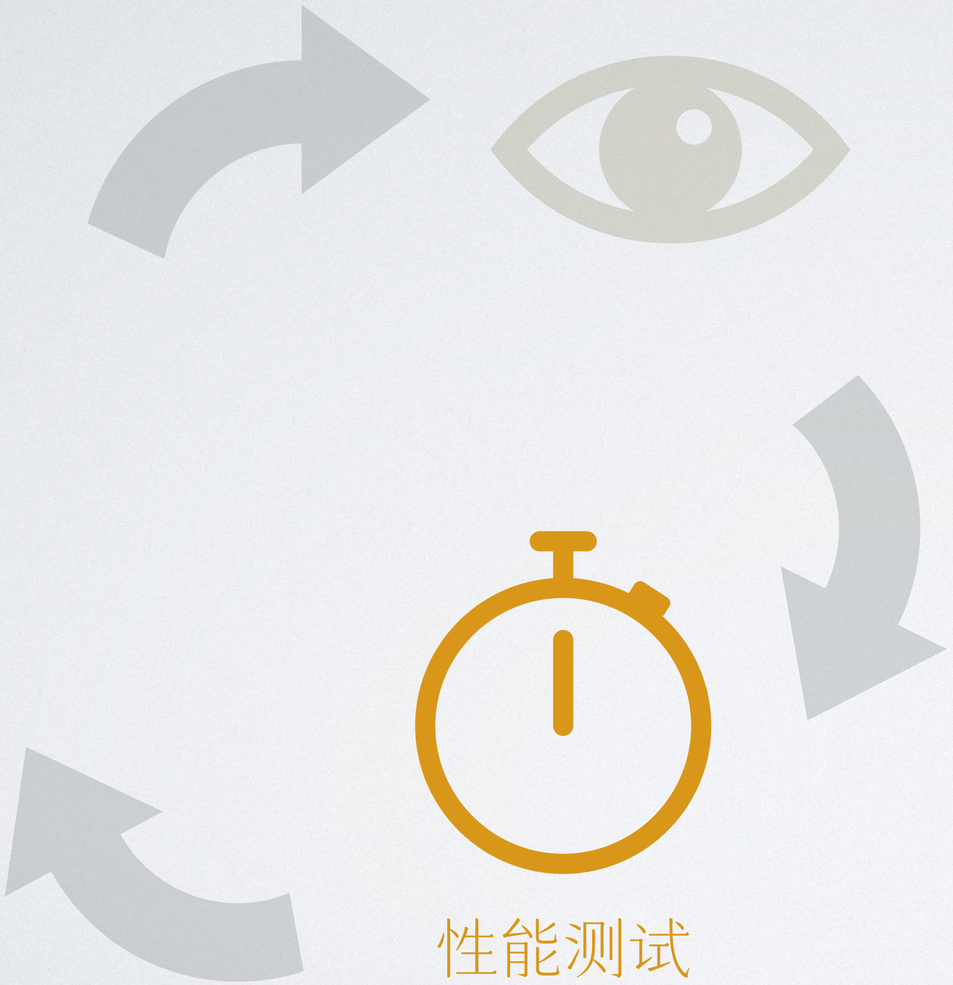
资源分析的时机



资源分析的时机



资源分析的时机



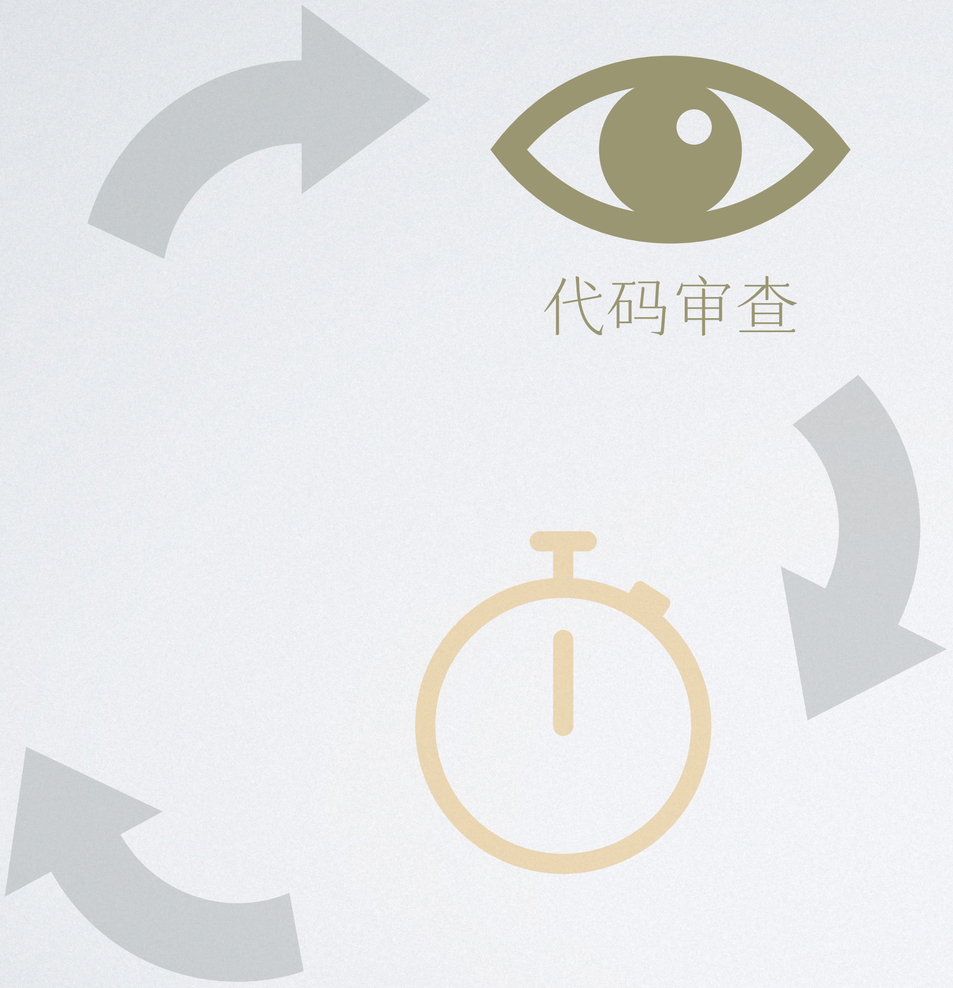
性能测试的优点：

- 实用性较强，对语言基本没要求
- 可以得到具体的测试用例

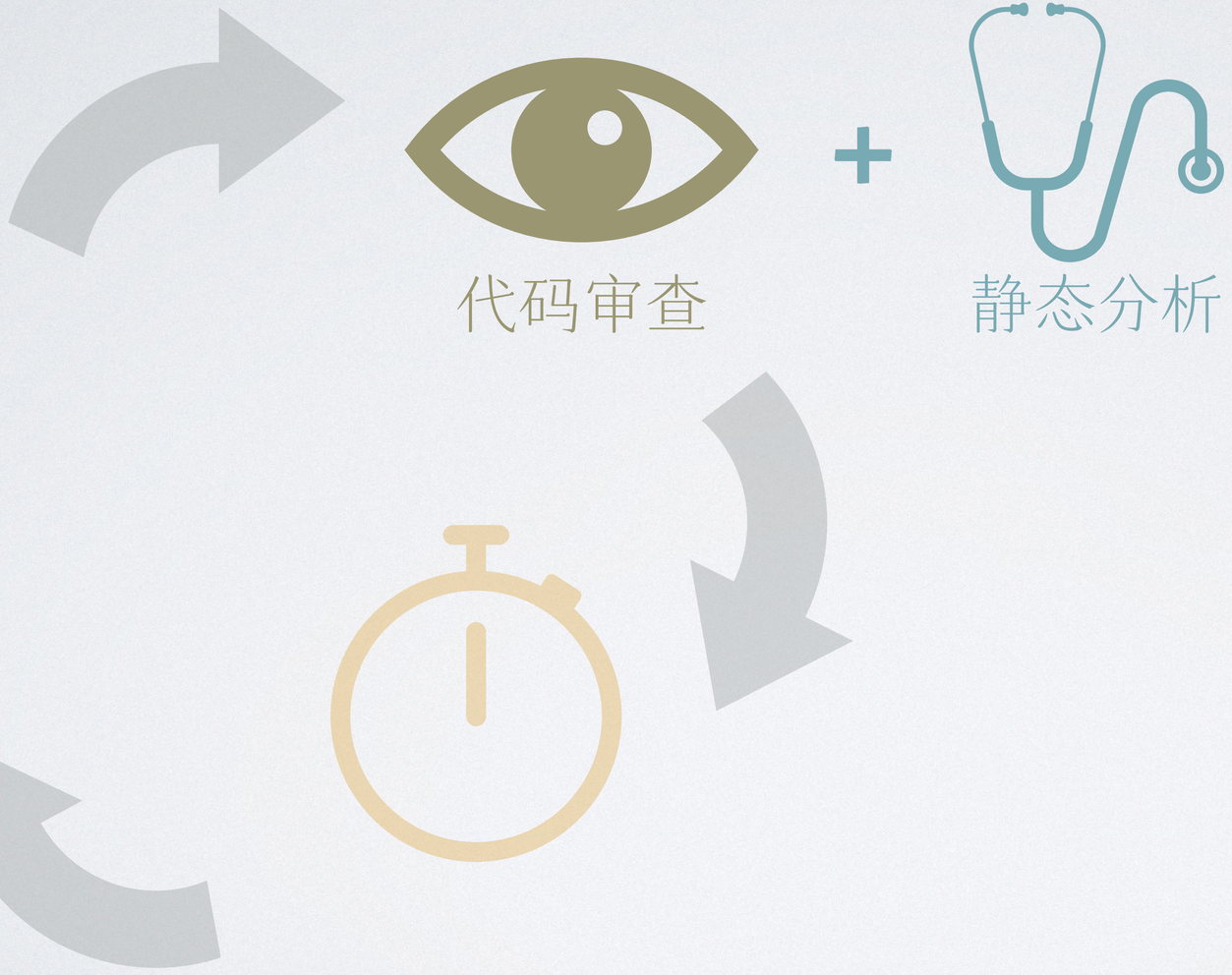
性能测试的缺点：

- 测试覆盖率不全
- 测试会进行较长的时间

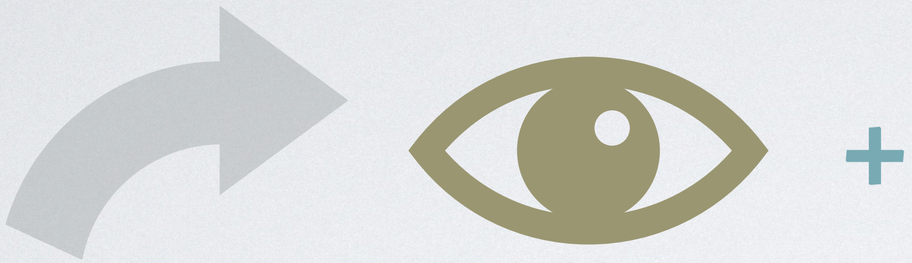
资源分析的时机



资源分析的时机



资源分析的时机



代码审查

静态分析

在编译期间分析
程序的资源消耗

静态分析的优点：

- 分析结果考虑了所有可能的输入情况
- 可以通过模块化提高分析效率

静态分析的缺点：

- 需要语言提供支持，对语言特性有要求
- 通常只能求出上界，可能导致假阳性

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

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

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

```java
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

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

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

```java
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
  }
}
```

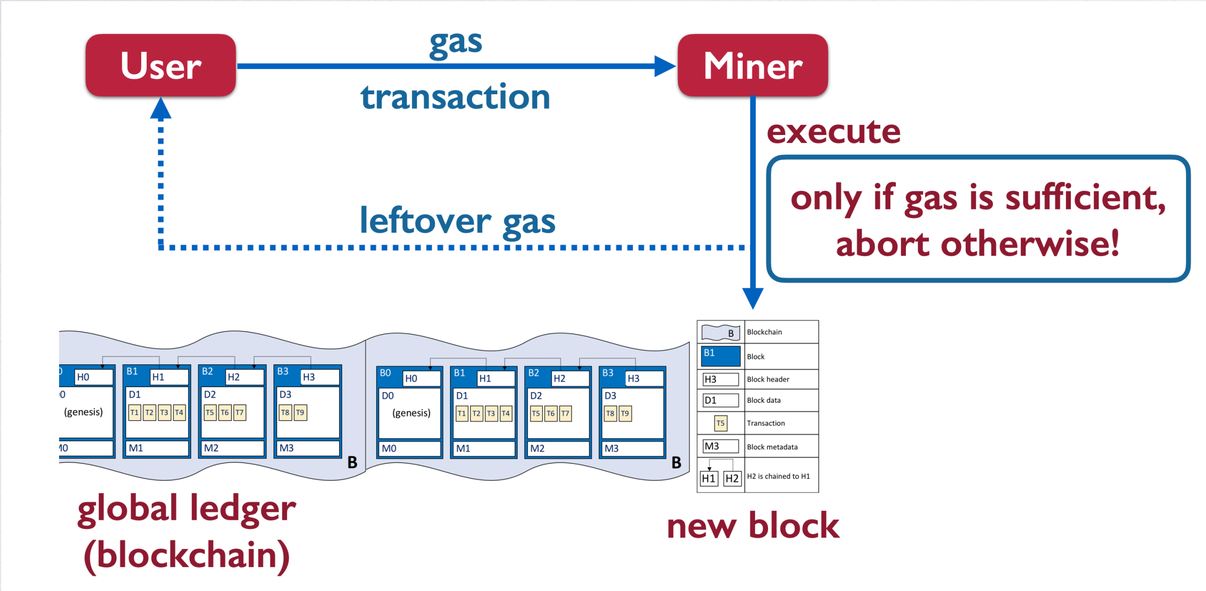
$8|list| + 16 = O(|list|)$

时间复杂度变高了！

$O(|list|^2)$

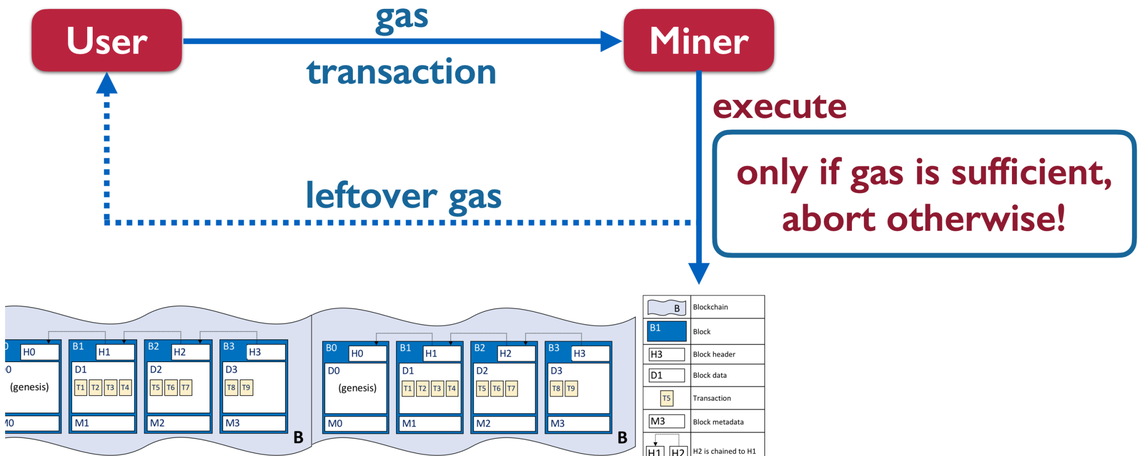
该案例来源于静态分析工具 Infer 的官方文档：https://fbinfer.com/docs/next/checker-cost/.

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.

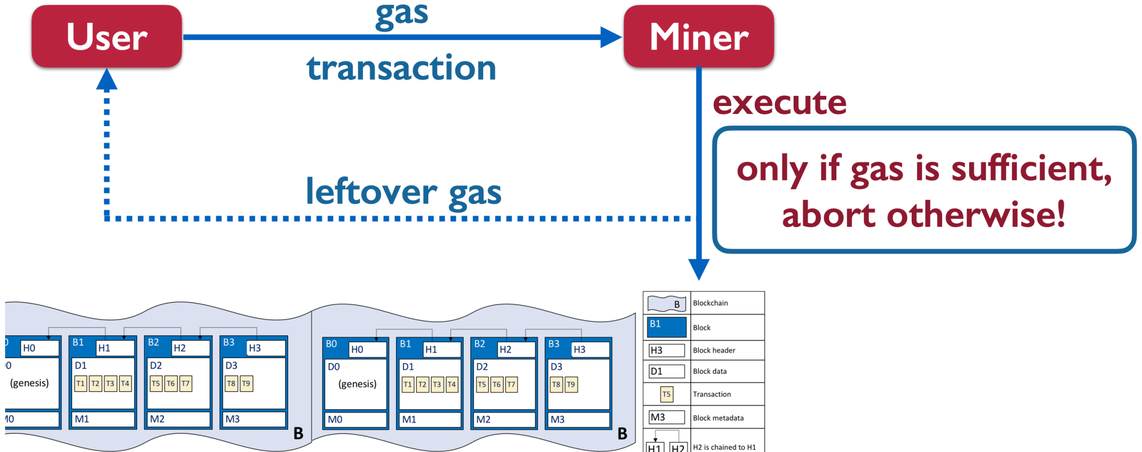
预测智能合约的燃气消耗




```
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.

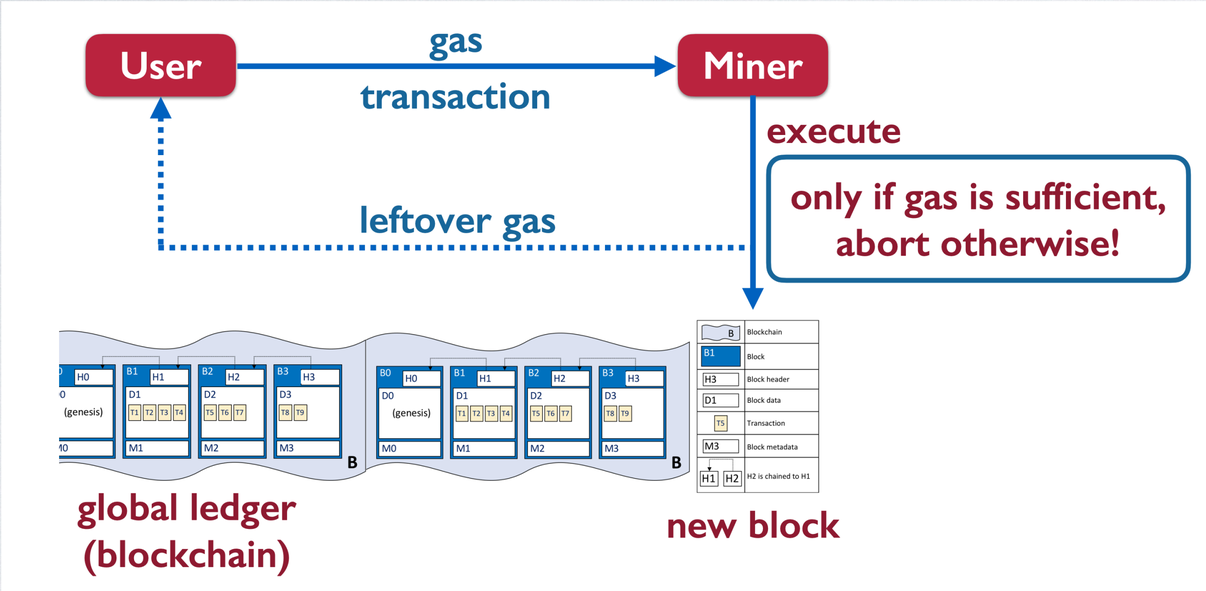
预测智能合约的燃气消耗




```
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



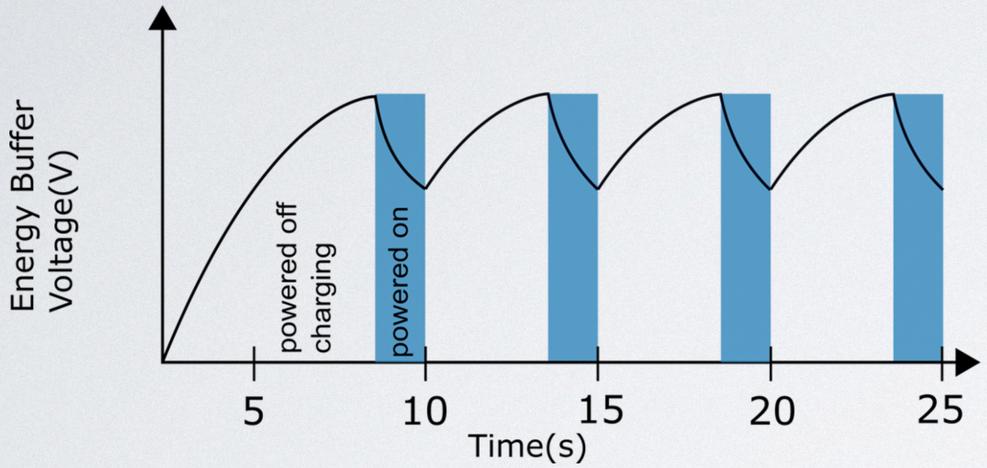

```
resource GasBalance {
  balance : Coin,
  gas : Gas(65)        // utilized to pay interest periodically
}
resource Bank {
  nogas_accounts : Map<address, Coin>,
  gas_accounts : Map<address, GasBalance>
}
fn [201] recharge(bank : &Bank)
fn [29] payInterest(bank : &Bank)
fn [34] signup(bank : &Bank,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

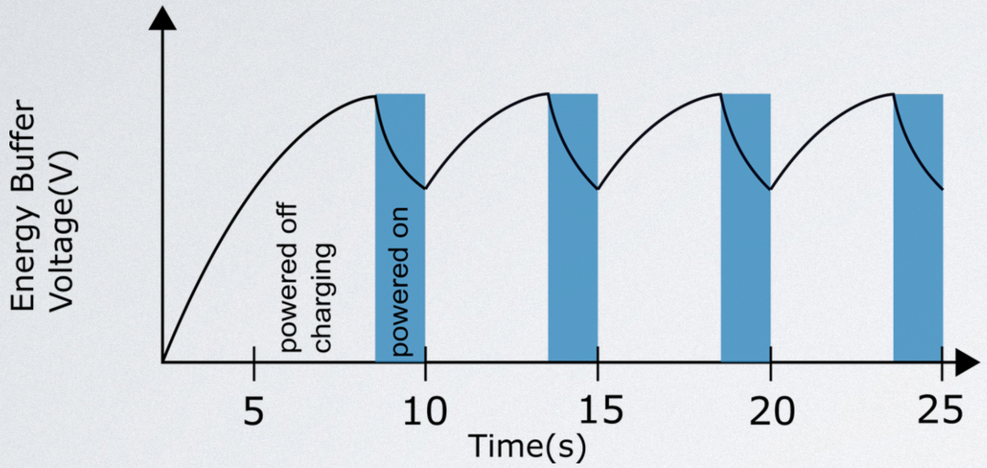
从燃气消耗到更物理的资源度量



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

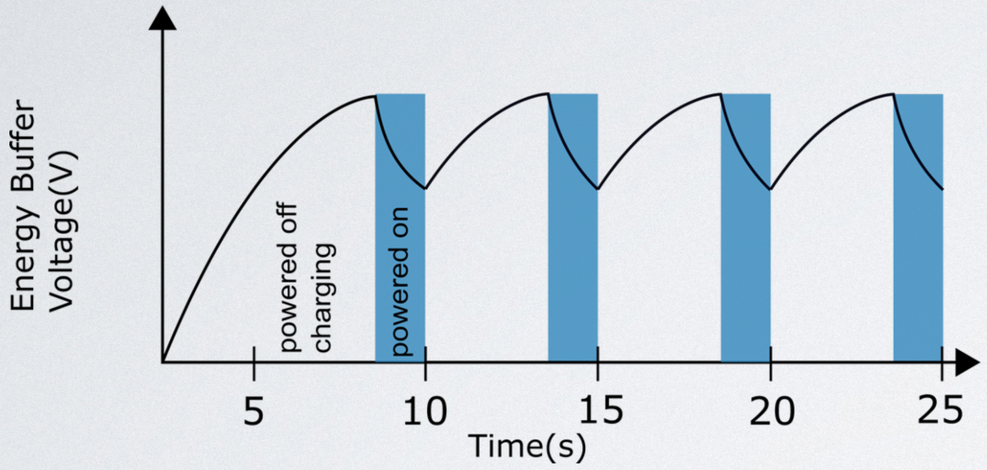


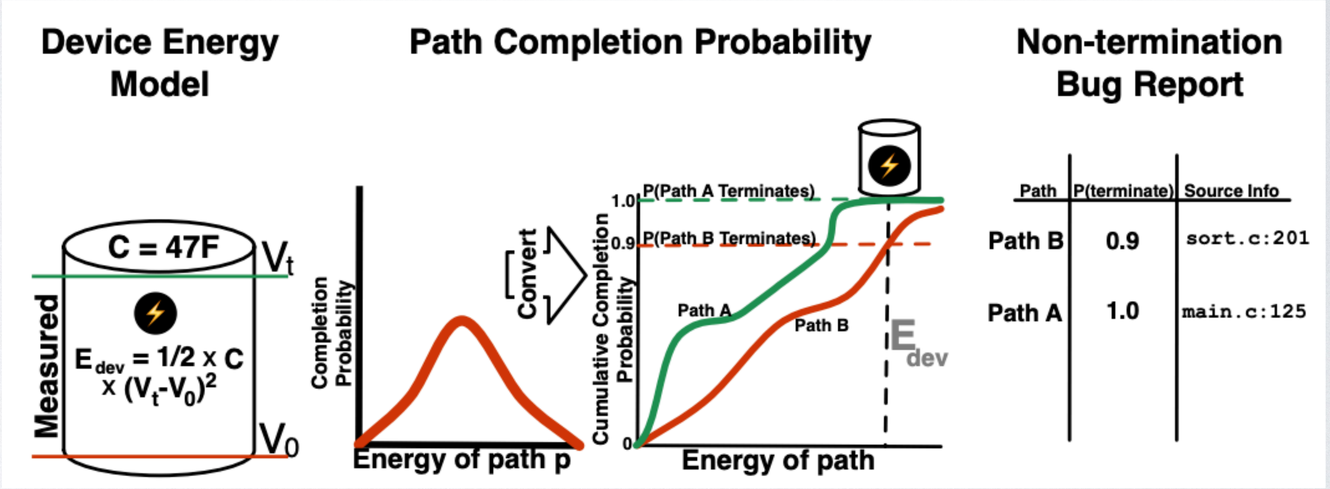
一类物联网设备
充电与计算间歇进行

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



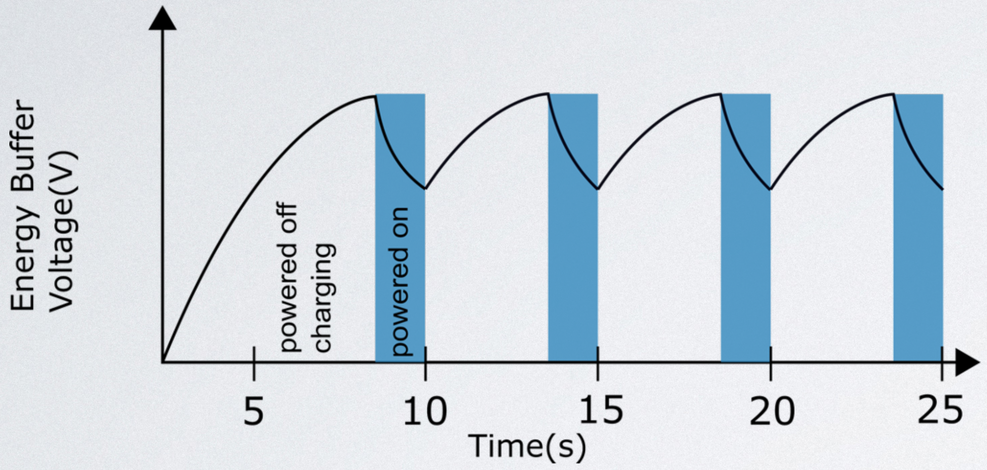


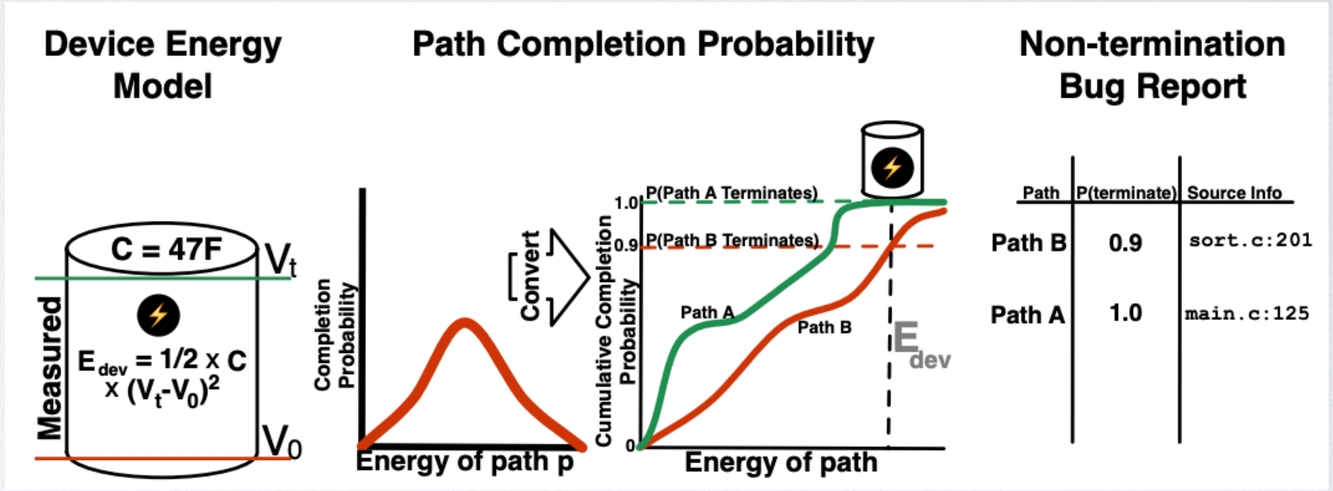
一类物联网设备
充电与计算间歇进行

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





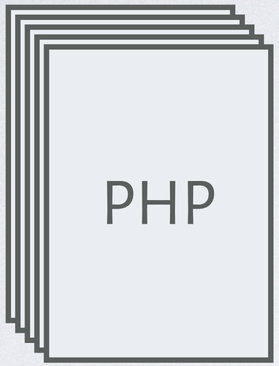
一类物联网设备
充电与计算间歇进行

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

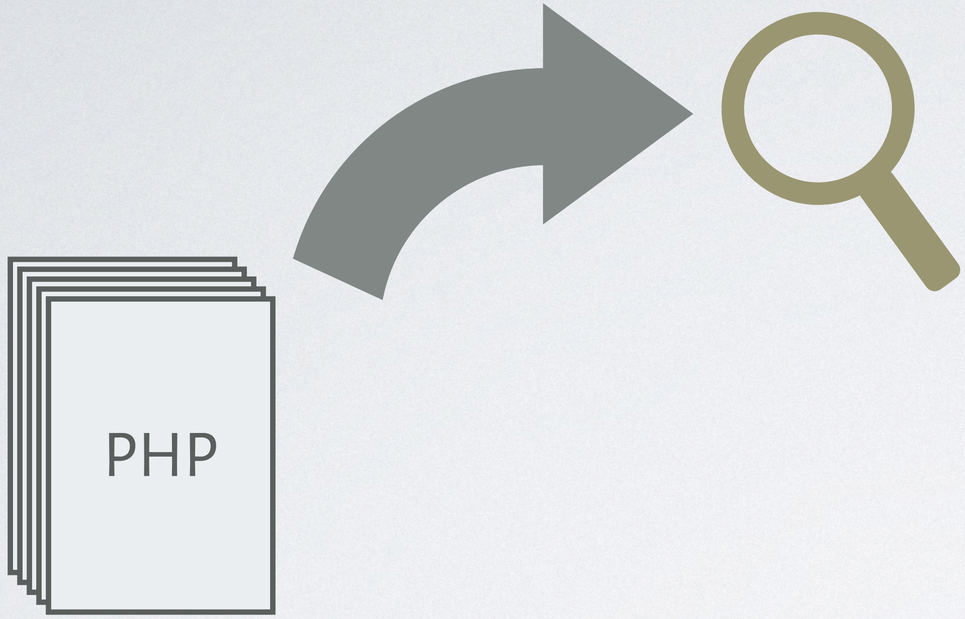
图片来自 CC 2018 的论文：A. Colin and B. Lucia, Termination Checking and Task Decomposition for Task-Based Intermittent Programs.

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.

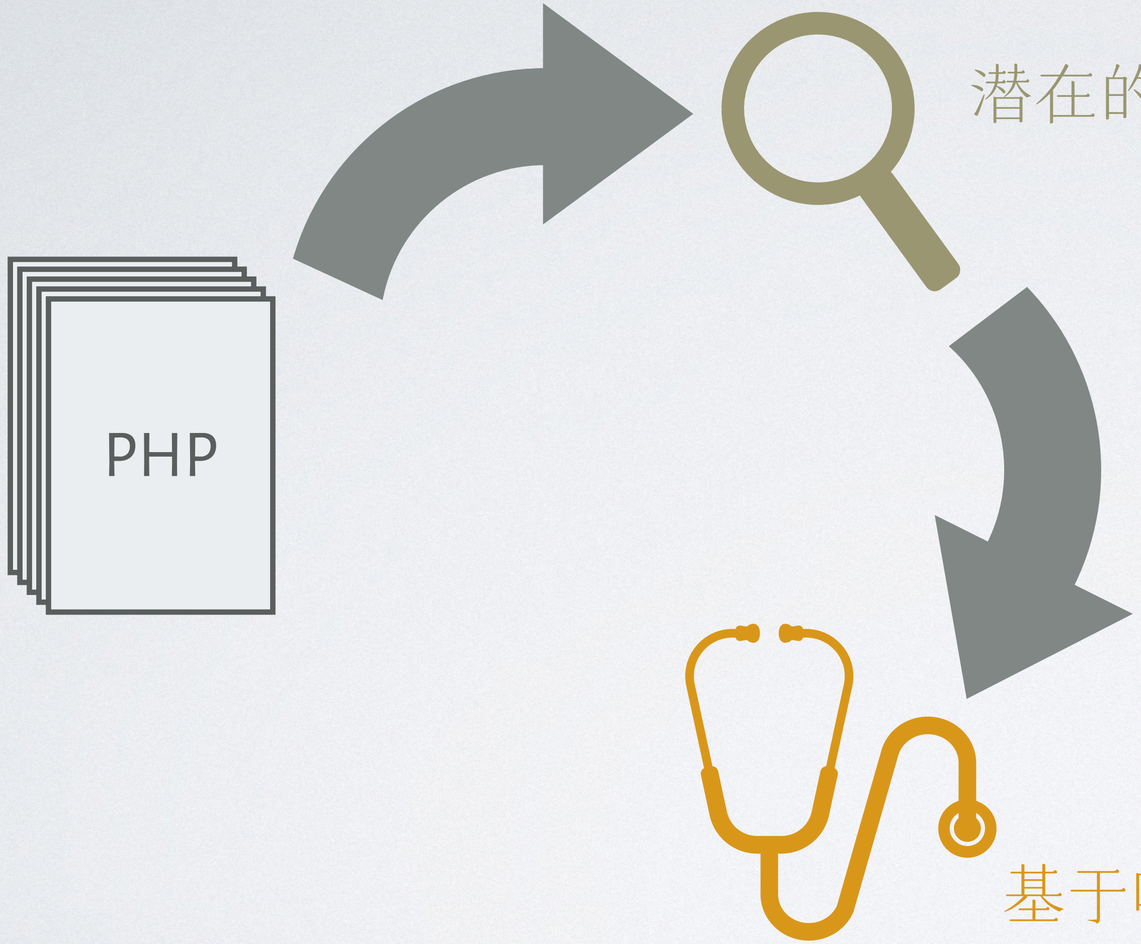
发现软件的拒绝服务漏洞



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

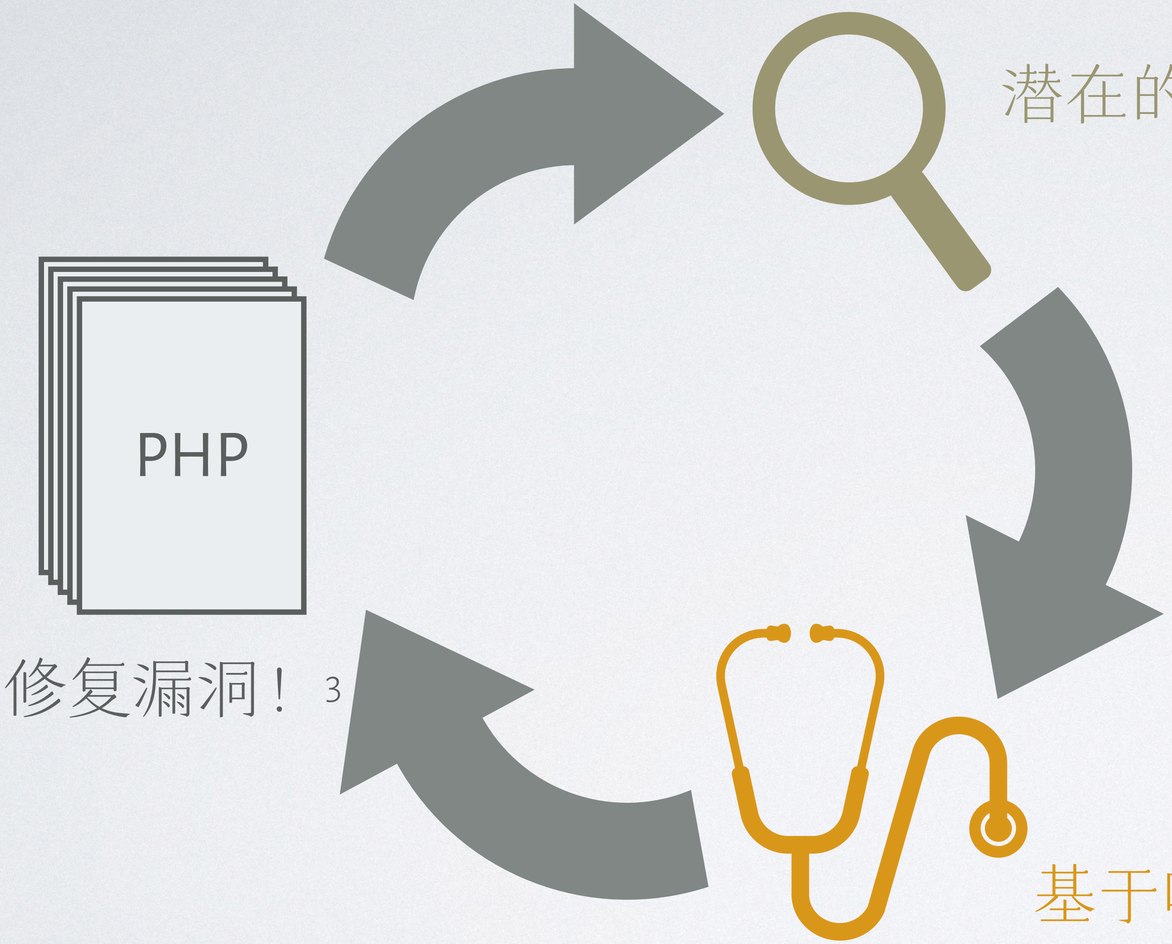


潜在的拒绝服务漏洞 [1]

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

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.

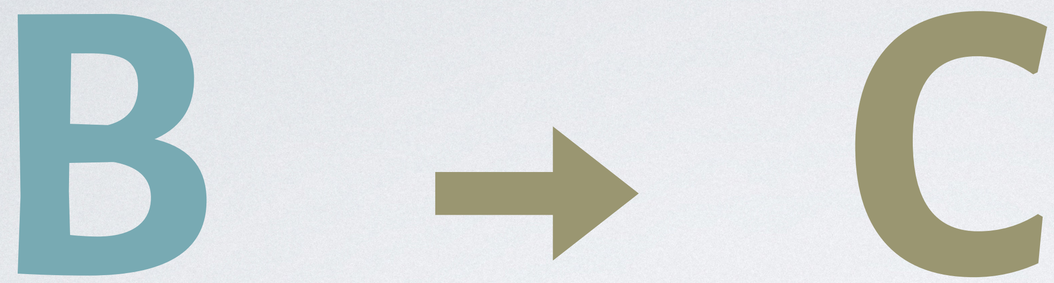
资源敏感的编程语言

资源敏感的编程语言

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！

资源敏感的编程语言



- 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！

资源敏感的编程语言

B → **C** → **Rust**

- 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！

资源敏感的编程语言

B → **C** → **Rust**

- 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！

资源敏感的编程语言

B → **C** → **Rust**

- 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！

资源敏感的编程语言

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

资源敏感的编程语言

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

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**

资源敏感的编程语言

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

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**
- 性能保障包含三方面：

资源敏感的编程语言

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

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**
- 性能保障包含三方面：

算法复杂度符合设计

资源敏感的编程语言

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

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**
- 性能保障包含三方面：

算法复杂度符合设计 **物理资源消耗满足预期**

资源敏感的编程语言

B → **C** → **Rust**

无安全保障	类型安全	类型、内存、并发安全
非常慢	很快啊	挺快的

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**
- 性能保障包含三方面：

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**

资源敏感的编程语言

B → **C** → **Rust**

无安全保障	类型安全	类型、内存、并发安全
非常慢	很快啊	挺快的

- 现代编程语言不仅要提供**安全保障**，还要提供**性能保障**
- 性能保障包含三方面：

算法复杂度符合设计

没有资源相关的安全漏洞

梦开始的地方：基于类型的资源分析

OCaml

RaML

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

[HDW17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

梦开始的地方：基于类型的资源分析

OCaml

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

RaML

append : $\langle L^9(\alpha) \times L^0(\alpha), 3 \rangle \to \langle L^0(\alpha), 0 \rangle$

[HDW17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

梦开始的地方：基于类型的资源分析

OCaml

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

RaML

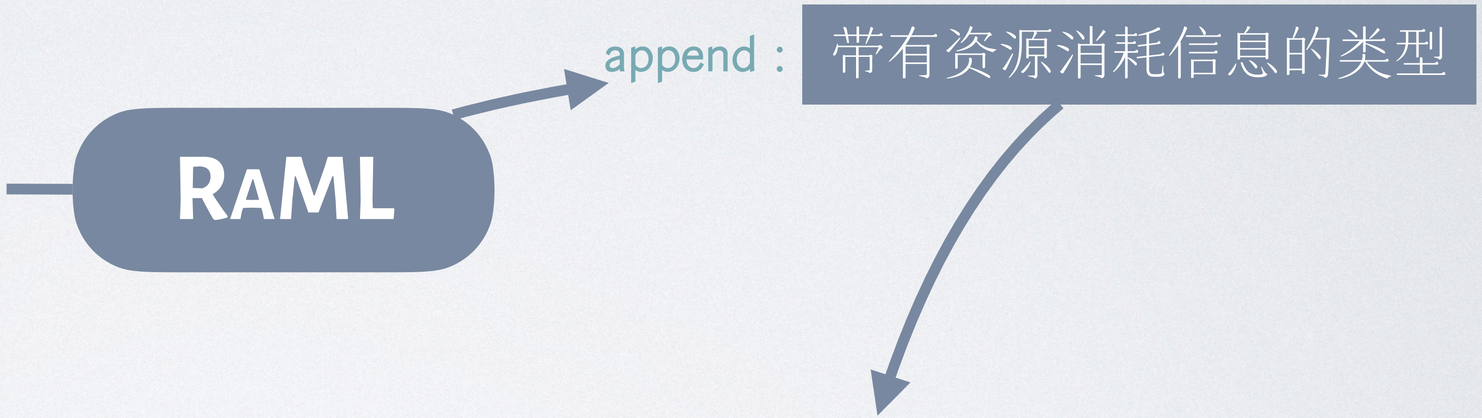
append : 带有资源消耗信息的类型

[HDW17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

梦开始的地方：基于类型的资源分析

OCaml

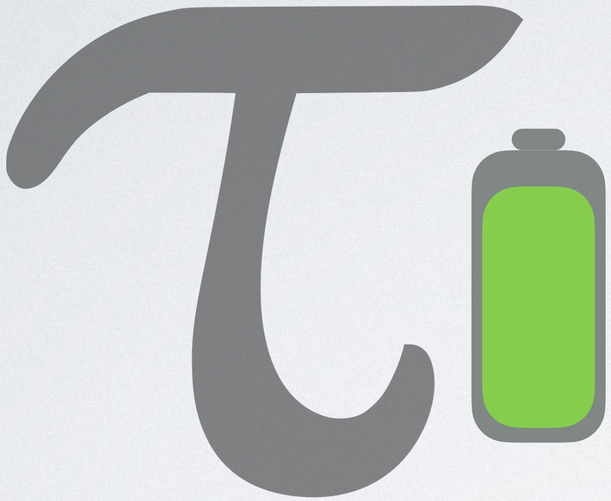
```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x::xs -> x::(append xs l2)
```



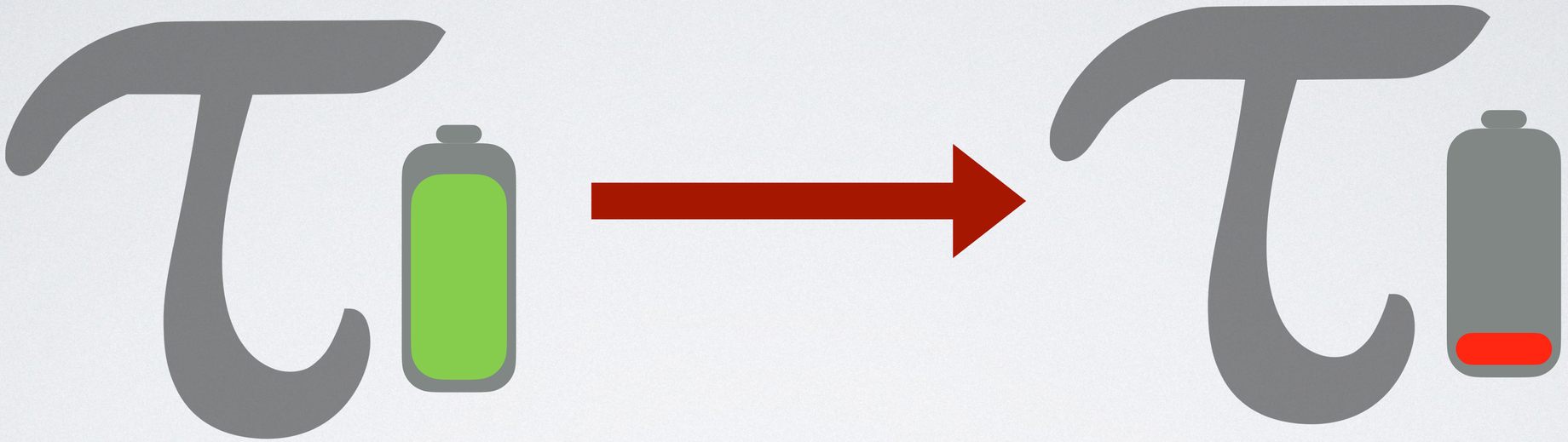
[HDW17] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

梦开始的地方：基于类型的资源分析

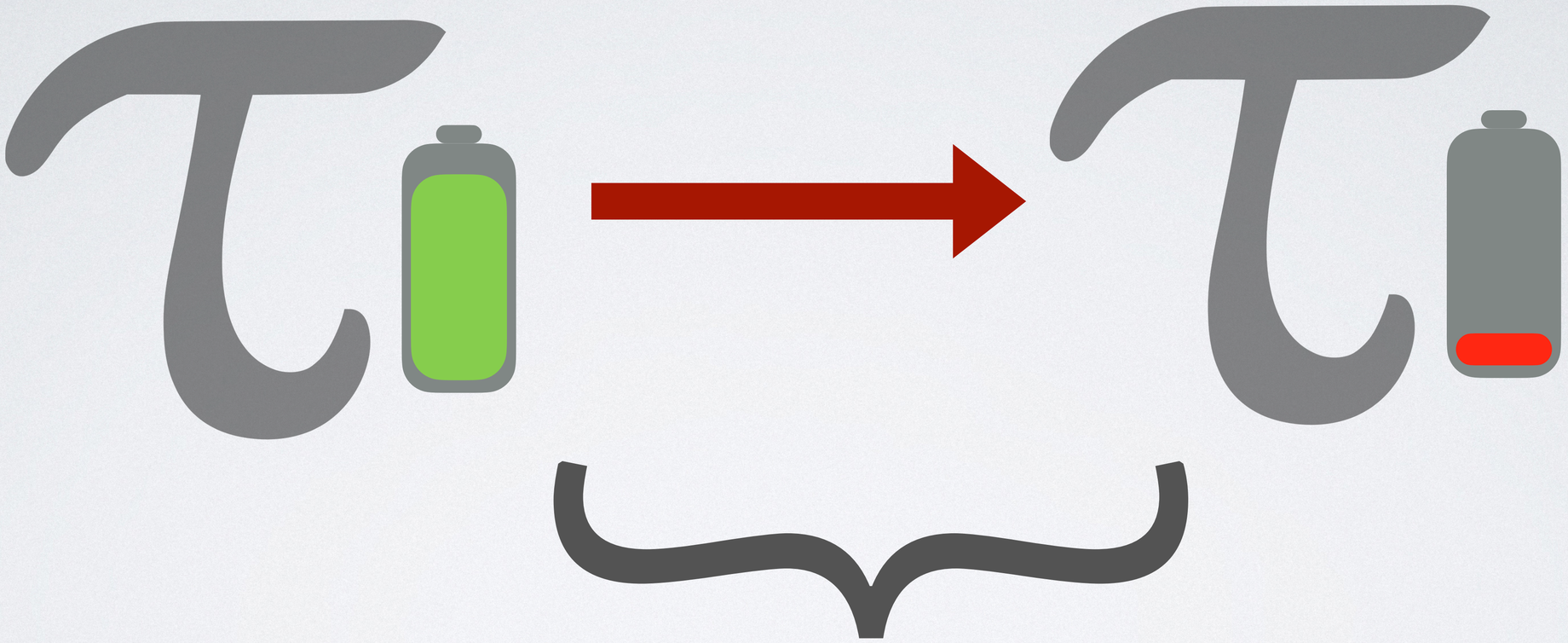
梦开始的地方：基于类型的资源分析



梦开始的地方：基于类型的资源分析



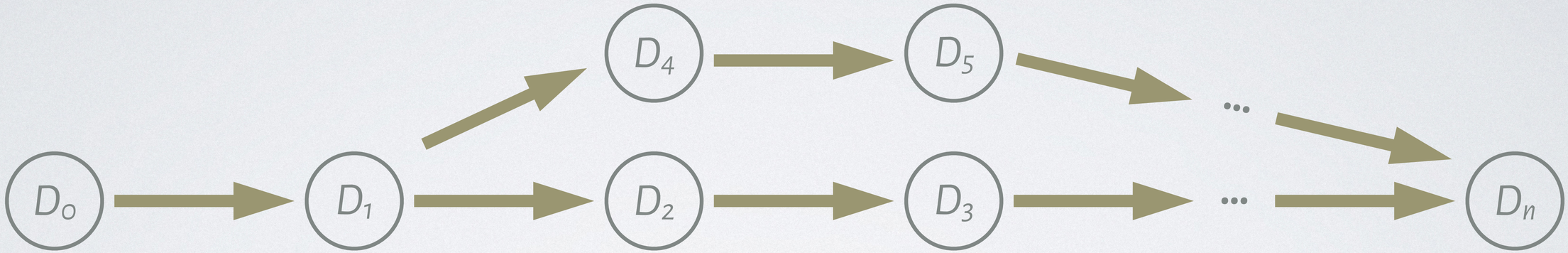
梦开始的地方：基于类型的资源分析



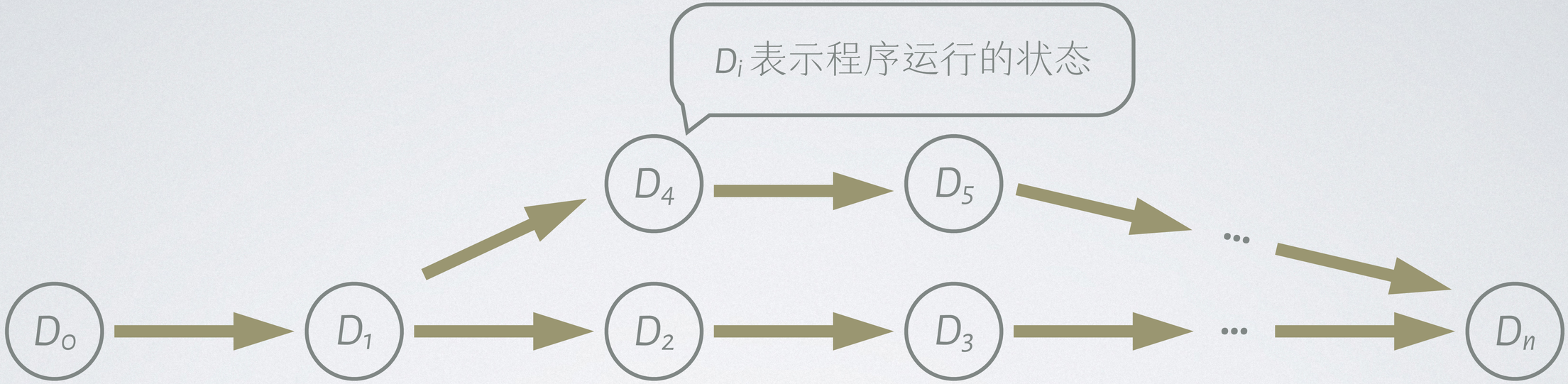
资源消耗

均摊分析的势能方法

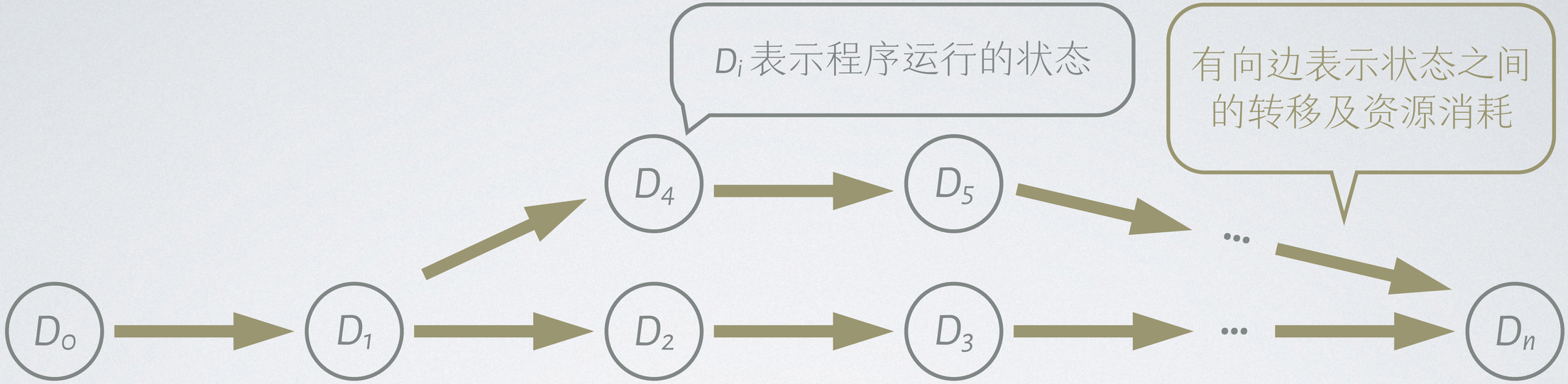
均摊分析的势能方法



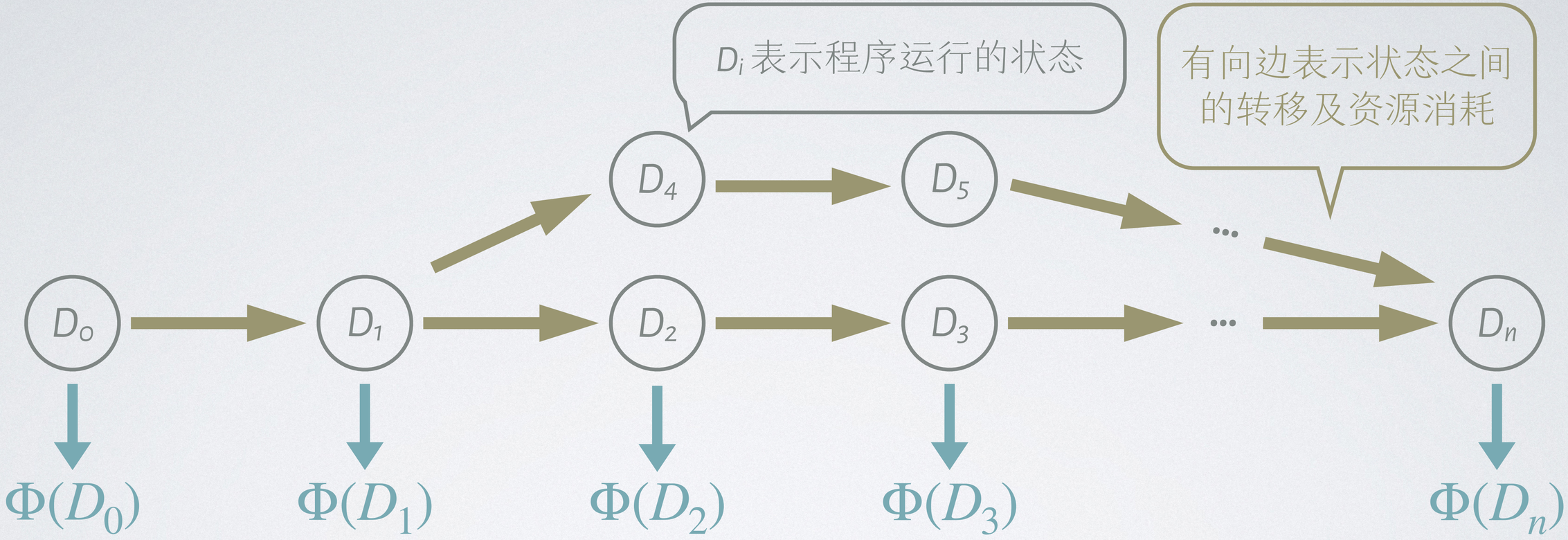
均摊分析的势能方法



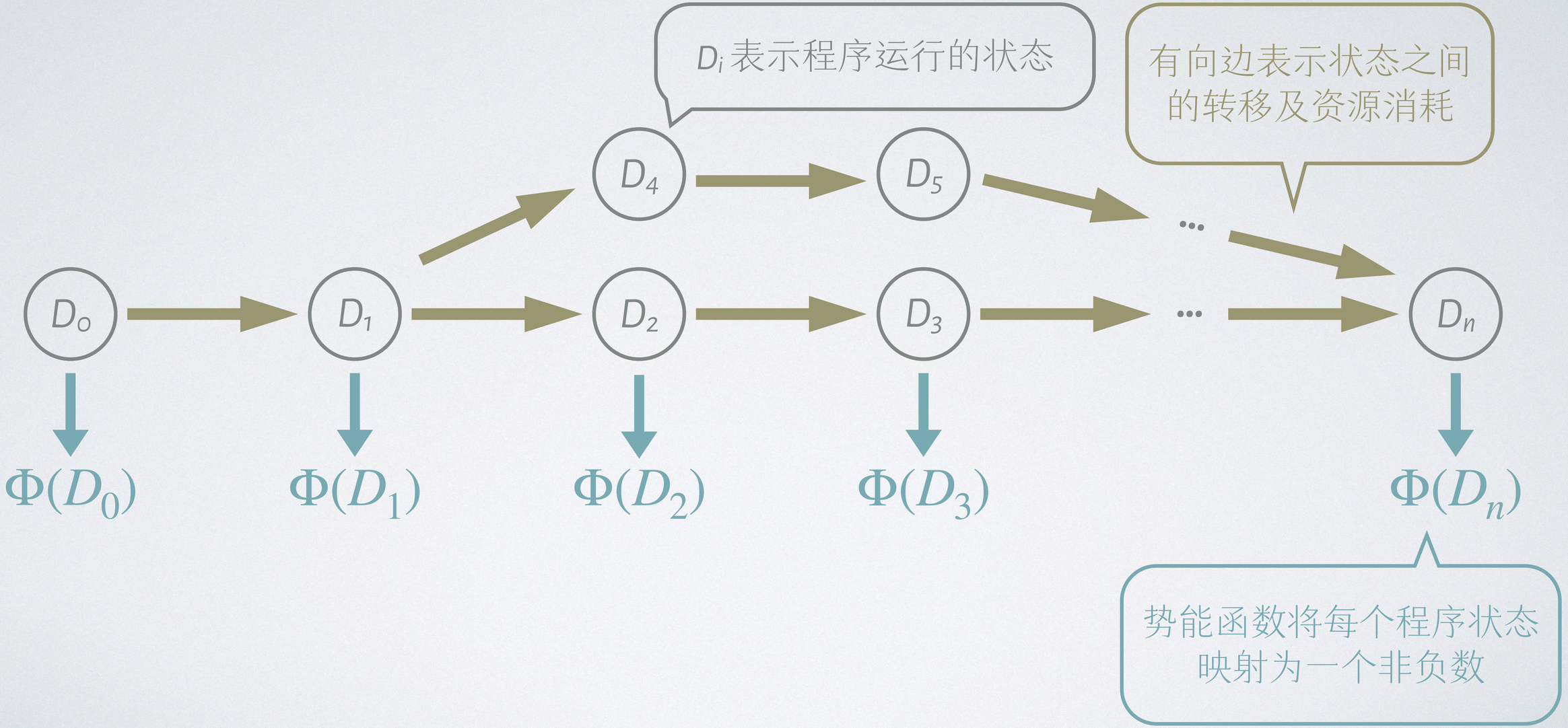
均摊分析的势能方法



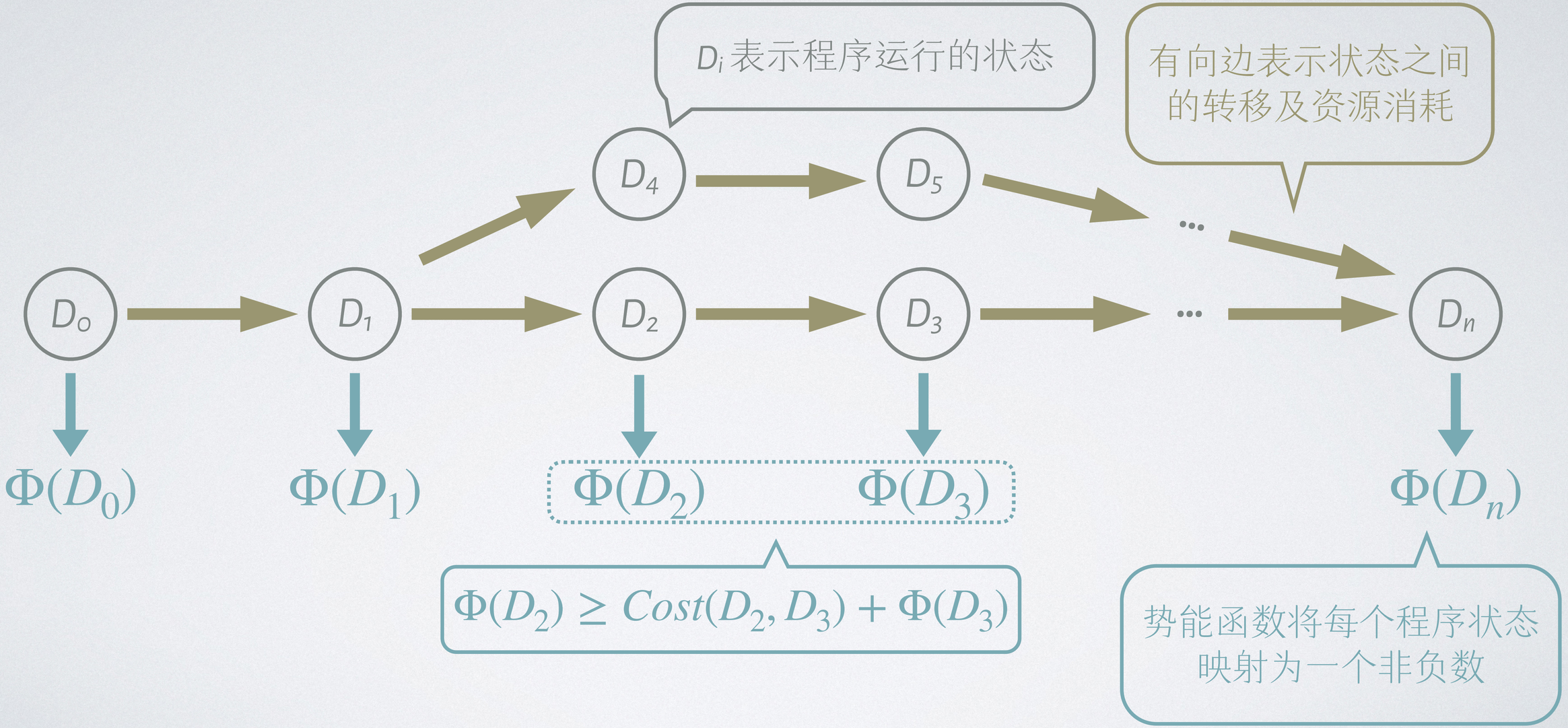
均摊分析的势能方法


D_i 表示程序运行的状态
有向边表示状态之间的转移及资源消耗
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$

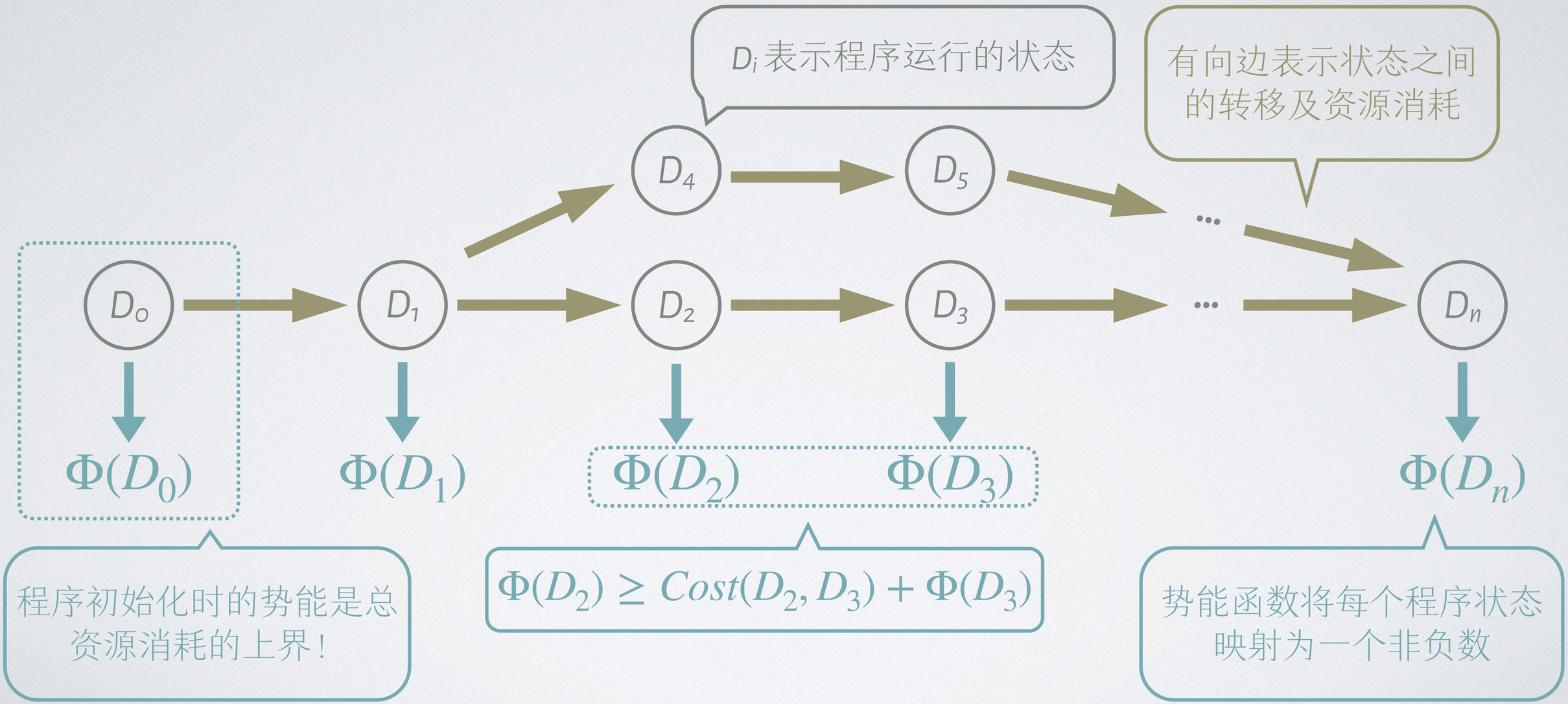
均摊分析的势能方法


D_i 表示程序运行的状态
有向边表示状态之间的转移及资源消耗
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
势能函数将每个程序状态映射为一个非负数

均摊分析的势能方法


D_i 表示程序运行的状态
有向边表示状态之间的转移及资源消耗
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
$\Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3)$
势能函数将每个程序状态映射为一个非负数

均摊分析的势能方法


D_i 表示程序运行的状态
有向边表示状态之间的转移及资源消耗
D_0
D_1
D_2
D_3
D_4
D_5
...
...
D_n
$\Phi(D_0)$
$\Phi(D_1)$
$\Phi(D_2)$
$\Phi(D_3)$
$\Phi(D_n)$
程序初始化时的势能是总资源消耗的上界！
$\Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3)$
势能函数将每个程序状态映射为一个非负数

带有势能标注的类型

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

带有势能标注的类型

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

通过 tick 显式标注程序的资源消耗模型

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

Lᵖ(a)

列表中的每个元素都携带了 p 单位的势能

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

Lᵖ(a)

列表中的每个元素都携带了 p 单位的势能

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

Lᵖ(a)

列表中的每个元素都携带了 p 单位的势能

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
```

带有势能标注的类型

$Cost = |\ell_1|$

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

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

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

通过 tick 显式标注程序的资源消耗模型

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
```

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
```

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
```

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
```

带有势能标注的类型

$Cost = |\ell_1|$

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

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

通过 tick 显式标注程序的资源消耗模型

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
[l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
```

带有势能标注的类型

$Cost = |\ell_1|$

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

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

通过 tick 显式标注程序的资源消耗模型

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

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
[l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
[x: a, rest: L⁰(a)]; 0 units
```

带有势能标注的类型

$Cost = |\ell_1|$

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

通过 tick 显式标注程序的资源消耗模型

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

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
[l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
[x: a, rest: L⁰(a)]; 0 units
// x 和 rest 被消耗且返回类型符合签名
```

带有势能标注的类型

$Cost = |\ell_1|$

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

$L^p(a)$

列表中的每个元素都携带了 p 单位的势能

通过 tick 显式标注程序的资源消耗模型

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

```
[l1: L¹(a), l2: L⁰(a)]; 0 units
// l1 被消耗
[l2: L⁰(a)]; 0 units
// l2 被消耗且返回类型符合签名
[l2: L⁰(a), x: a, xs: L¹(a)]; 1 unit
[l2: L⁰(a), x: a, xs: L¹(a)]; 0 units
[x: a, rest: L⁰(a)]; 0 units
// x 和 rest 被消耗且返回类型符合签名
```

原理：每个程序点的**势能函数**由程序操作的数据结构的**静态类型标注**所决定

基于线性规划的类型推导

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

基于线性规划的类型推导

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

线性约束

p≥0, q≥0, r≥0, s≥0, t≥0

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lp(a), l2: Lq(a)]; r units
```

线性约束

p≥0, q≥0, r≥0, s≥0, t≥0

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lᵖ(a), l2: Lᵠ(a)]; r units
// l1 被消耗
```

线性约束

p≥0, q≥0, r≥0, s≥0, t≥0

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lᵖ(a), l2: Lᵠ(a)]; r units
// l1 被消耗
[l2: Lᵠ(a)]; r units
```

线性约束

$p \geq 0, q \geq 0, r \geq 0, s \geq 0, t \geq 0$

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lᵖ(a), l2: Lᵠ(a)]; r units
// l1 被消耗
[l2: Lᵠ(a)]; r units
// l2 被消耗且返回类型符合签名
```

线性约束

$p \geq 0, q \geq 0, r \geq 0, s \geq 0, t \geq 0$

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lᵖ(a), l2: Lᵠ(a)]; r units
// l1 被消耗
[l2: Lᵠ(a)]; r units
// l2 被消耗且返回类型符合签名
```

线性约束

$p \geq 0, q \geq 0, r \geq 0, s \geq 0, t \geq 0$

$q \geq s, r \geq t$

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lp(a), l2: Lq(a)]; r units
// l1 被消耗
[l2: Lq(a)]; r units
// l2 被消耗且返回类型符合签名
[l2: Lq(a), x: a, xs: Lp(a)]; r+p units
```

线性约束

$p \geq 0, q \geq 0, r \geq 0, s \geq 0, t \geq 0$

$q \geq s, r \geq t$

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lp(a), l2: Lq(a)]; r units
// l1 被消耗
[l2: Lq(a)]; r units
// l2 被消耗且返回类型符合签名
[l2: Lq(a), x: a, xs: Lp(a)]; r+p units
[l2: Lq(a), x: a, xs: Lp(a)]; r+p-1 units
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t
```

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lp(a), l2: Lq(a)]; r units
// l1 被消耗
[l2: Lq(a)]; r units
// l2 被消耗且返回类型符合签名
[l2: Lq(a), x: a, xs: Lp(a)]; r+p units
[l2: Lq(a), x: a, xs: Lp(a)]; r+p-1 units
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t

r+p-1≥0
```

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

```
let rec append l1 l2 =                [l1: Lᵖ(a), l2: Lᵠ(a)]; r units
  match l1 with                       // l1 被消耗
  | [] ->                             [l2: Lᵠ(a)]; r units
    l2                                // l2 被消耗且返回类型符合签名
  | x::xs ->                          [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p units
    let () = tick(1) in               [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p-1 units
    let rest = append xs l2 in        [x: a, rest: Lˢ(a)]; p-1+t units
    x::rest
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t

r+p-1≥0
```

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

```
let rec append l1 l2 =                [l1: Lᵖ(a), l2: Lᵠ(a)]; r units
  match l1 with                       // l1 被消耗
  | [] ->                             [l2: Lᵠ(a)]; r units
    l2                                // l2 被消耗且返回类型符合签名
  | x::xs ->                          [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p units
    let () = tick(1) in               [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p-1 units
    let rest = append xs l2 in        [x: a, rest: Lˢ(a)]; p-1+t units
    x::rest
```

线性约束

$p \geq 0, q \geq 0, r \geq 0, s \geq 0, t \geq 0$

$q \geq s, r \geq t$

$r+p-1 \geq 0$

$p \geq p, q \geq q, r+p-1 \geq r$

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

```
let rec append l1 l2 =                [l1: Lᵖ(a), l2: Lᵠ(a)]; r units
  match l1 with                       // l1 被消耗
  | [] ->                             [l2: Lᵠ(a)]; r units
    l2                                // l2 被消耗且返回类型符合签名
  | x::xs ->                          [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p units
    let () = tick(1) in               [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p-1 units
    let rest = append xs l2 in        [x: a, rest: Lˢ(a)]; p-1+t units
    x::rest                           // x 和 rest 被消耗且返回类型符合签名
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t

r+p-1≥0
p≥p, q≥q, r+p-1≥r
```

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

```
let rec append l1 l2 =                [l1: Lᵖ(a), l2: Lᵠ(a)]; r units
  match l1 with                       // l1 被消耗
  | [] ->                             [l2: Lᵠ(a)]; r units
    l2                                // l2 被消耗且返回类型符合签名
  | x::xs ->                          [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p units
    let () = tick(1) in               [l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p-1 units
    let rest = append xs l2 in        [x: a, rest: Lˢ(a)]; p-1+t units
    x::rest                           // x 和 rest 被消耗且返回类型符合签名
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t

r+p-1≥0
p≥p, q≥q, r+p-1≥r
p-1+t≥s+t
```

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lᵖ(a), l2: Lᵠ(a)]; r units
// l1 被消耗
[l2: Lᵠ(a)]; r units
// l2 被消耗且返回类型符合签名
[l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p units
[l2: Lᵠ(a), x: a, xs: Lᵖ(a)]; r+p-1 units
[x: a, rest: Lˢ(a)]; p-1+t units
// x 和 rest 被消耗且返回类型符合签名
```

线性约束

```
p≥0,q≥0,r≥0,s≥0,t≥0

q≥s,r≥t

r+p-1≥0
p≥p,q≥q,r+p-1≥r
p-1+t≥s+t
```

p=1, q=r=s=t=0

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

```
let rec append l1 l2 =                [l1: Lp(a), l2: Lq(a)]; r units
  match l1 with                       // l1 被消耗
  | [] ->                             [l2: Lq(a)]; r units
    l2                                // l2 被消耗且返回类型符合签名
  | x::xs ->                          [l2: Lq(a), x: a, xs: Lp(a)]; r+p units
    let () = tick(1) in               [l2: Lq(a), x: a, xs: Lp(a)]; r+p-1 units
    let rest = append xs l2 in        [x: a, rest: Ls(a)]; p-1+t units
    x::rest                           // x 和 rest 被消耗且返回类型符合签名
```

线性约束

```
p≥0, q≥0, r≥0, s≥0, t≥0

q≥s, r≥t

r+p-1≥0
p≥p, q≥q, r+p-1≥r
p-1+t≥s+t
```

append : $\langle L^1(\alpha) \times L^0(\alpha), 0\rangle \to \langle L^0(\alpha), 0\rangle$ ← p=1, q=r=s=t=0

基于线性规划的类型推导

p,q,r,s,t 是未知数值量

append : $\langle L^p(\alpha) \times L^q(\alpha), r\rangle \to \langle L^s(\alpha), t\rangle$

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

```
[l1: Lp(a), l2: Lq(a)]; r units
// l1 被消耗
[l2: Lq(a)]; r units
// l2 被消耗且返回类型符合签名
[l2: Lq(a), x: a, xs: Lp(a)]; r+p units
[l2: Lq(a), x: a, xs: Lp(a)]; r+p-1 units
[x: a, rest: Ls(a)]; p-1+t units
// x 和 rest 被消耗且返回类型符合签名
```

线性约束

```
p≥0,q≥0,r≥0,s≥0,t≥0

q≥s,r≥t

r+p-1≥0
p≥p,q≥q,r+p-1≥r
p-1+t≥s+t
```

append : $\langle L^2(\alpha) \times L^1(\alpha), 3\rangle \to \langle L^1(\alpha), 3\rangle$ ← `p=2, q=s=1, r=t=3`

基于势能方法的自动资源分析

[JHL+10]	线性形式的资源消耗上界，高阶函数，参数多态
[HDW17]	多元多项式形式的资源消耗上界，高阶函数，归纳数据类型
[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
[KWP+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
[WKH20]	平均情况资源消耗分析，概率程序期望资源消耗分析
[KWR+20]	支持多种形式资源消耗上界的精化类型系统
[GKH23]	正则递归类型

[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*.
[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*.
[KWP+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*.
[WKH20] D. Wang, D. M. Kahn, and J. Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types. In *ICFP'20*.
[KWR+20] T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. 2020. Liquid Resource Types. In *ICFP'20*.
[GKH23] J. Grosen, D. M. Kahn, and J. Hoffmann. 2023. Automatic Amortized Resource Analysis with Regular Recursive Types. In *LICS'23*.

源程序资源消耗复杂度分析

算法复杂度符合设计

源程序资源消耗复杂度分析

算法复杂度符合设计

- 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

算法复杂度符合设计

- 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
  - 从研发新型语言的视角看，设计类型系统使得资源分析更容易、更精确

源程序资源消耗复杂度分析

算法复杂度符合设计

- 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
  - 从研发新型语言的视角看，设计类型系统使得资源分析更容易、更精确

工作1：基于势能分析的 Rust 程序资源分析

Q. Lian and D. Wang. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. *Working Paper*.

[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.

工作1：基于势能分析的 Rust 程序资源分析

Q. Lian and D. Wang. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. *Working Paper*.


```
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.

工作1：基于势能分析的 Rust 程序资源分析

Q. Lian and D. Wang. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. *Working Paper*.


```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.

工作1：基于势能分析的 Rust 程序资源分析

Q. Lian and D. Wang. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. *Working Paper*.


```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.

工作1：基于势能分析的 Rust 程序资源分析

Q. Lian and D. Wang. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. *Working Paper*.

[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.

工作1：基于势能分析的 Rust 程序资源分析

工作1：基于势能分析的 Rust 程序资源分析

- 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

工作1：基于势能分析的 Rust 程序资源分析

- 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

工作1：基于势能分析的 Rust 程序资源分析

- **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;
```

工作1：基于势能分析的 Rust 程序资源分析

- 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;
```

工作1：基于势能分析的 Rust 程序资源分析

- 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

工作1：基于势能分析的 Rust 程序资源分析

- 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**

工作1：基于势能分析的 Rust 程序资源分析

```rust
fn iter_twice(l: &List) {
  // l : &list(4)
  iter(&*l); // share 4 as 2 + 2, &*l : &list(2)
  // l : &list(2)
  iter(&*l); // share 2 as 2 + 0, &*l : &list(2)
  // l : &list(0)
}
```

(a) Shared Reborrowing

```rust
fn update(l: &mut List) {
  iter(&*l);
  // l : &mut list(0)
  *l = Cons(3, Box::new(Nil));
  // l : &mut list(4)
  iter(&*l); iter(&*l);
}
```

(b) Mutating A Mutable Borrow

```rust
fn prophecy() {
  let mut l = Cons(3, Box::new(Nil));
  // l : list(p)
  let x = &mut l;
  // l : list(q),  x : &mut(list(p ), list(q))
  update(x);    // x : &mut list(p'), list(q))
  /* drop(x) */ // p' >= q
}
```

(c) Creating & Dropping A Mutable Borrow

```rust
fn weak(b: bool, l1: &mut List, l2: &mut List) {
  let l = if b {
    &mut *l1 // : &mut(list(p1), list(q1))
  } else {
    &mut *l2 // : &mut(list(p2), list(q2))
  }; // : &mut(list(min(p1,p2)), list(max(q1,q2)))
  update(l);
}
```

(d) Mutable Reborrowing & Aliasing

工作1：基于势能分析的 Rust 程序资源分析

进行 shared borrow 时分割已有势能

```
fn iter_twice(l: &List) {
  // l : &list(4)
  iter(&*l); // share 4 as 2 + 2, &*l : &list(2)
  // l : &list(2)
  iter(&*l); // share 2 as 2 + 0, &*l : &list(2)
  // l : &list(0)
}
```

(a) Shared Reborrowing

```
fn update(l: &mut List) {
  iter(&*l);
  // l : &mut list(0)
  *l = Cons(3, Box::new(Nil));
  // l : &mut list(4)
  iter(&*l); iter(&*l);
}
```

(b) Mutating A Mutable Borrow

```
fn prophecy() {
  let mut l = Cons(3, Box::new(Nil));
  // l : list(p)
  let x = &mut l;
  // l : list(q),  x : &mut(list(p ), list(q))
  update(x);    // x : &mut list(p'), list(q))
  /* drop(x) */ // p' >= q
}
```

(c) Creating & Dropping A Mutable Borrow

```
fn weak(b: bool, l1: &mut List, l2: &mut List) {
  let l = if b {
    &mut *l1 // : &mut(list(p1), list(q1))
  } else {
    &mut *l2 // : &mut(list(p2), list(q2))
  }; // : &mut(list(min(p1,p2)), list(max(q1,q2)))
  update(l);
}
```

(d) Mutable Reborrowing & Aliasing

工作1：基于势能分析的 Rust 程序资源分析

进行 shared borrow 时分割已有势能

```
fn iter_twice(l: &List) {
  // l : &list(4)
  iter(&*l); // share 4 as 2 + 2, &*l : &list(2)
  // l : &list(2)
  iter(&*l); // share 2 as 2 + 0, &*l : &list(2)
  // l : &list(0)
}
```

(a) Shared Reborrowing

```
fn update(l: &mut List) {
  iter(&*l);
  // l : &mut list(0)
  *l = Cons(3, Box::new(Nil));
  // l : &mut list(4)
  iter(&*l); iter(&*l);
}
```

(b) Mutating A Mutable Borrow

修改 mutable borrow 时直接改写其势能

```
fn prophecy() {
  let mut l = Cons(3, Box::new(Nil));
  // l : list(p)
  let x = &mut l;
  // l : list(q),  x : &mut(list(p ), list(q))
  update(x);    // x : &mut list(p'), list(q))
  /* drop(x) */ // p' >= q
}
```

(c) Creating & Dropping A Mutable Borrow

```
fn weak(b: bool, l1: &mut List, l2: &mut List) {
  let l = if b {
    &mut *l1 // : &mut(list(p1), list(q1))
  } else {
    &mut *l2 // : &mut(list(p2), list(q2))
  }; // : &mut(list(min(p1,p2)), list(max(q1,q2)))
  update(l);
}
```

(d) Mutable Reborrowing & Aliasing

工作1：基于势能分析的 Rust 程序资源分析

进行 shared borrow 时分割已有势能

```
fn iter_twice(l: &List) {
  // l : &list(4)
  iter(&*l); // share 4 as 2 + 2, &*l : &list(2)
  // l : &list(2)
  iter(&*l); // share 2 as 2 + 0, &*l : &list(2)
  // l : &list(0)
}
```

(a) Shared Reborrowing

```
fn update(l: &mut List) {
  iter(&*l);
  // l : &mut list(0)
  *l = Cons(3, Box::new(Nil));
  // l : &mut list(4)
  iter(&*l); iter(&*l);
}
```

修改 mutable borrow 时直接改写其势能

(b) Mutating A Mutable Borrow

进行 mutable borrow 时引入 prophecy：&mut(L(p), L(q)) 表示当前势能为 L(p)，生命周期结束时预期为 L(q)

```
fn prophecy() {
  let mut l = Cons(3, Box::new(Nil));
  // l : list(p)
  let x = &mut l;
  // l : list(q),  x : &mut(list(p ), list(q))
  update(x);    // x : &mut list(p'), list(q))
  /* drop(x) */ // p' >= q
}
```

(c) Creating & Dropping A Mutable Borrow

```
fn weak(b: bool, l1: &mut List, l2: &mut List) {
  let l = if b {
    &mut *l1 // : &mut(list(p1), list(q1))
  } else {
    &mut *l2 // : &mut(list(p2), list(q2))
  }; // : &mut(list(min(p1,p2)), list(max(q1,q2)))
  update(l);
}
```

(d) Mutable Reborrowing & Aliasing

工作1：基于势能分析的 Rust 程序资源分析

进行 shared borrow 时分割已有势能

```
fn iter_twice(l: &List) {
  // l : &list(4)
  iter(&*l); // share 4 as 2 + 2, &*l : &list(2)
  // l : &list(2)
  iter(&*l); // share 2 as 2 + 0, &*l : &list(2)
  // l : &list(0)
}
```

(a) Shared Reborrowing

```
fn update(l: &mut List) {
  iter(&*l);
  // l : &mut list(0)
  *l = Cons(3, Box::new(Nil));
  // l : &mut list(4)
  iter(&*l); iter(&*l);
}
```

修改 mutable borrow 时直接改写其势能

(b) Mutating A Mutable Borrow

进行 mutable borrow 时引入 prophecy：&mut(L(p), L(q)) 表示当前势能为 L(p)，生命周期结束时预期为 L(q)

```
fn prophecy() {
  let mut l = Cons(3, Box::new(Nil));
  // l : list(p)
  let x = &mut l;
  // l : list(q),  x : &mut(list(p ), list(q))
  update(x);    // x : &mut list(p'), list(q))
  /* drop(x) */ // p' >= q
}
```

(c) Creating & Dropping A Mutable Borrow

```
fn weak(b: bool, l1: &mut List, l2: &mut List) {
  let l = if b {
    &mut *l1 // : &mut(list(p1), list(q1))
  } else {
    &mut *l2 // : &mut(list(p2), list(q2))
  }; // : &mut(list(min(p1,p2)), list(max(q1,q2)))
  update(l);
}
```

对 weak update 需要做保守分析

工作 2：基于势能的资源分析 + 精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.

T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.

H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

工作 2：基于势能的资源分析 + 精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.
T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.
H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

- 最开始虽然是程序合成的工作，但其核心是一个**带资源标注类型**的函数式语言

工作 2：基于势能的资源分析 + 精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.
T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.
H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

- 最开始虽然是程序合成的工作，但其核心是一个**带资源标注类型**的函数式语言
- 程序合成的部分是归约为**类型制导的程序合成**问题

工作2：基于势能的资源分析+精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.
T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.
H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

- 最开始虽然是程序合成的工作，但其核心是一个**带资源标注类型**的函数式语言
- 程序合成的部分是归约为**类型制导的程序合成**问题

```
  rep : (n: int) -> a ->
{ v: List a | len(v) = n }
```

工作2：基于势能的资源分析+精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.
T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.
H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

- 最开始虽然是程序合成的工作，但其核心是一个**带资源标注类型**的函数式语言
- 程序合成的部分是归约为**类型制导的程序合成**问题

```
  rep : (n: int) -> a ->
{ v: List a | len(v) = n }

let rec rep n x =
  if n <= 0
  then []
  else x::(rep (n - 1) x)
```

工作2：基于势能的资源分析+精化类型系统

T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *PLDI'19*.
T. Knoth, D. Wang, A. Reynolds, J. Hoffmann, and N. Polikarpova. Liquid Resource Types. In ICFP'20.
H. Xu and D. Wang. Depentently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs. *Working Paper*.

- 最开始虽然是程序合成的工作，但其核心是一个**带资源标注类型**的函数式语言
- 程序合成的部分是归约为**类型制导的程序合成**问题

```
  rep : (n: int) -> a ->
{ v: List a | len(v) = n }

let rec rep n x =
  if n <= 0
  then []
  else x::(rep (n - 1) x)
```

将程序合成问题转化为
搜索一个符合目标类型的程序

以 Refinement Types 为规约

以 Refinement Types 为规约

$\{ v : B \mid \Psi \}$

一个满足 Ψ 的具有类型 B 的值 v

以 Refinement Types 为规约

$\{\ v\colon B \mid \Psi\ \}$

一个满足 Ψ 的具有类型 B 的值 v

$\{\ v\colon \mathrm{Int} \mid v \geq 0\ \}$

一个非负整数

以 Refinement Types 为规约

`{ v: B | Ψ }` 一个满足 Ψ 的具有类型 B 的值 v

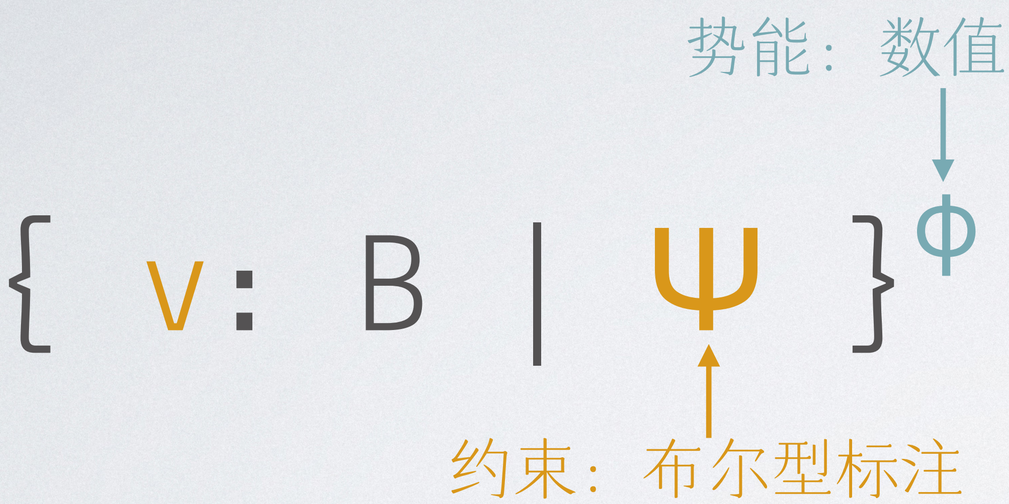
`{ v: Int | v ≥ 0 }` 一个非负整数

```
(xs: List a) -> { v: List a | len(v) = len(xs) + 1 }
```

一个函数，输入为一个列表 xs，输出为一个长度恰好比 xs 多 1 的列表

ReSyn： Refinement Types + 线性势能函数

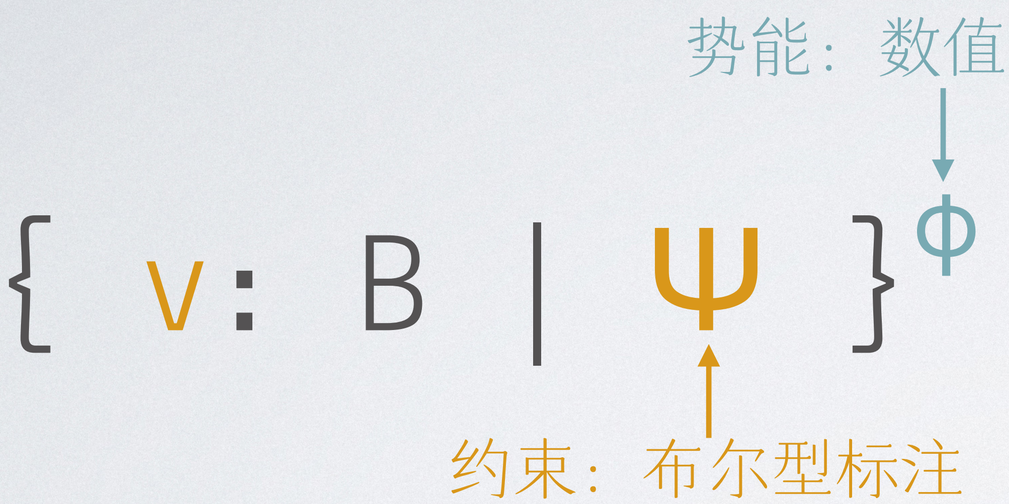
ReSyn： Refinement Types + 线性势能函数



$\{ v: \mathrm{Int} \mid v \geq 0 \}^{5 \cdot v}$

一个非负整数，携带了 5 倍于其自身值的势能

ReSyn： Refinement Types + 线性势能函数



$\{ v: Int \mid v \geq 0 \}^{5 \cdot v}$

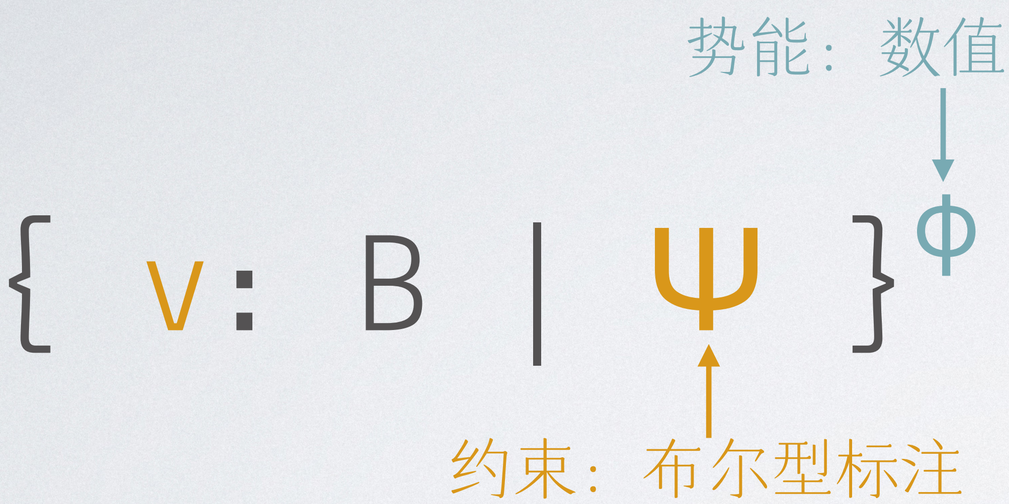
一个非负整数，携带了 5 倍于其自身值的势能

$List\ \{ v: a \mid true \}^{ite(v \geq 0, 1, 0)}$

一个列表，其中的每个非负元素
携带了 1 单位的势能

ite = if-then-else

ReSyn： Refinement Types + 线性势能函数



设计了将两种标注整合的类型系统和
基于该类型系统的程序合成算法

$\{ v: \mathrm{Int} \mid v \geq 0 \}^{5 \cdot v}$

一个非负整数，携带了 5 倍于其自身值的势能

$\mathrm{List}\ \{ v: a \mid \mathrm{true} \}^{\mathrm{ite}(v \geq 0, 1, 0)}$

一个列表，其中的每个非负元素
携带了 1 单位的势能

ite = if-then-else

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

资源制导的程序合成

```
common : (xs: SortedList a[1]) -> (ys: SortedList a[1]) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else x::(common xt ys)
```

平方时间复杂度

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else x::(common xt ys)
```

平方时间复杂度

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    match ys with
    | [] -> []
    | y::yt ->
      if x < y then common xt ys
      else if y < x then common xs yt
      else x::(common xt yt)
```

ReSyn：线性时间复杂度

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  ??
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

```
ys: List aᵖ <: List a¹
ys: List aᵠ <: List a¹
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

```
ys: List aᵖ <: List a¹      [p ≥ 1]
ys: List aᵠ <: List a¹      [q ≥ 1]
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

势能分割

```
ys: List aᵖ <: List a¹      [p ≥ 1]
ys: List aᵠ <: List a¹      [q ≥ 1]
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

势能分割

```
ys: List aᵖ <: List a¹      [p ≥ 1]
ys: List aᵠ <: List a¹      [q ≥ 1]
```

```
[1 ≥ p+q]
```

资源制导的程序合成

```
common : (xs: SortedList a¹) -> (ys: SortedList a¹) ->
{ v: SortedList a | elems(v) = elems(xs) ∩ elems(ys) }
```

```
let rec common xs ys =
  match xs with
  | [] -> []
  | x::xt ->
    if not (member x ys)
    then common xt ys
    else ??
```

势能分割

```
ys: List aᵖ <: List a¹
ys: List aᵠ <: List a¹
```

```
[1 ≥ p+q]

[p ≥ 1]
[q ≥ 1]
```

约束不可能满足！

资源消耗相关的安全漏洞

没有资源相关的安全漏洞

资源消耗相关的安全漏洞

没有资源相关的安全漏洞

- **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

资源消耗相关的安全漏洞

没有资源相关的安全漏洞

- **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

工作 3：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*.

工作 3：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

工作 3：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

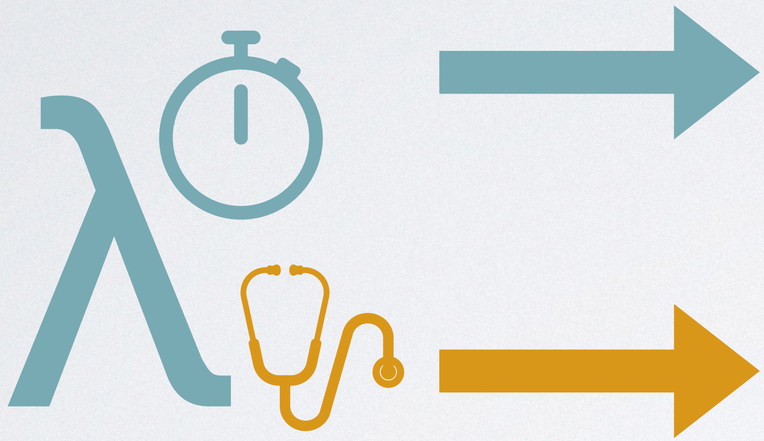
指导

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

工作 3：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

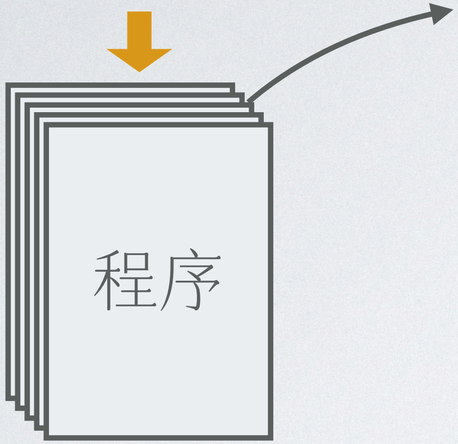
基于符号执行的最坏情况输入生成

基于符号执行的最坏情况输入生成

输入的规约
（例如列表长度）

基于符号执行的最坏情况输入生成

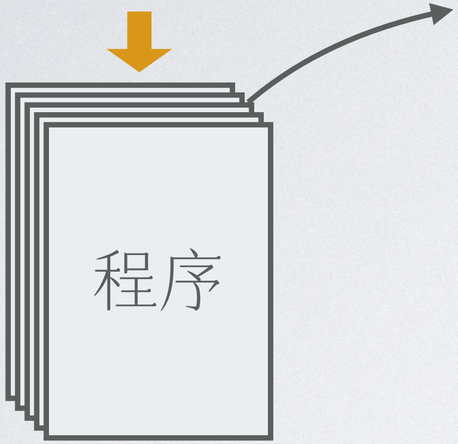
输入的规约
（例如列表长度）



可能的执行路径 1

基于符号执行的最坏情况输入生成

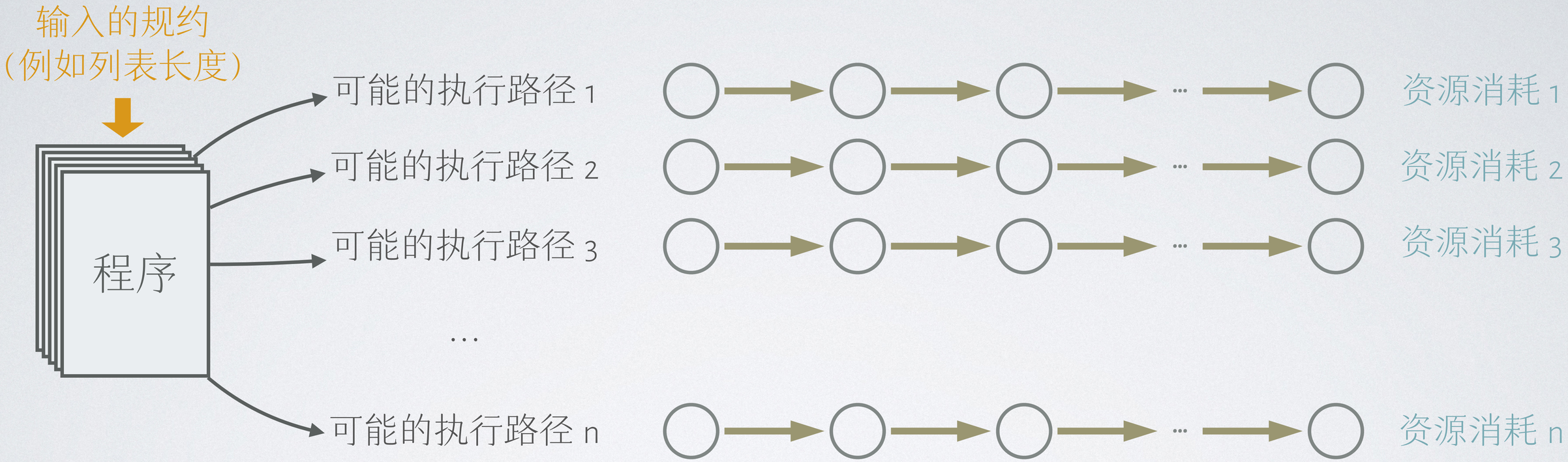
输入的规约
（例如列表长度）



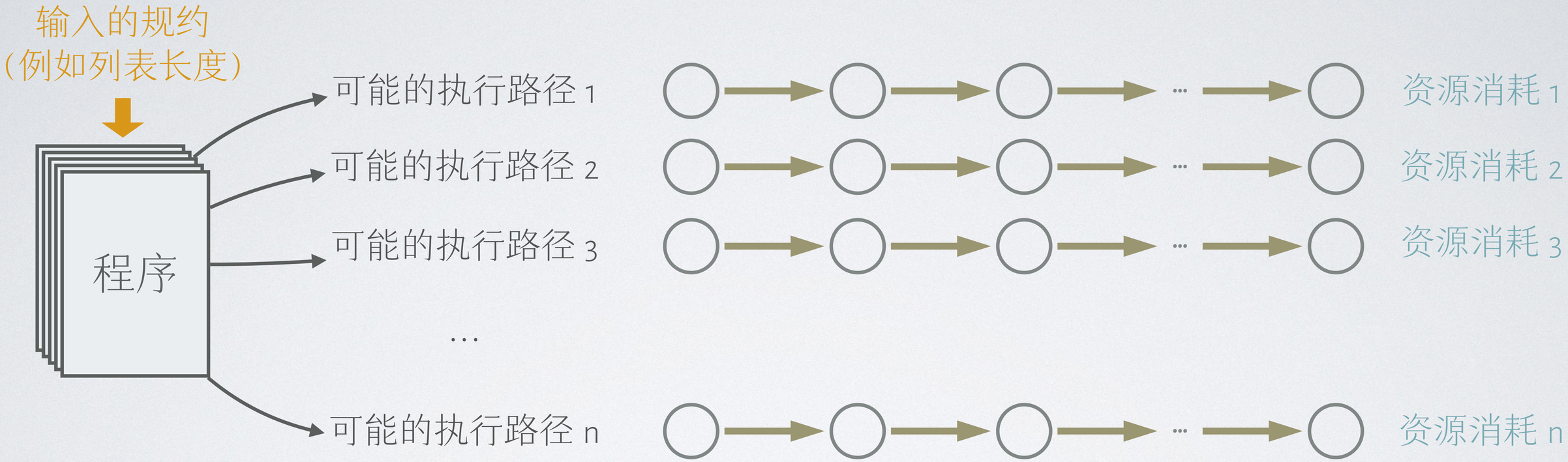
可能的执行路径 1

资源消耗 1

基于符号执行的最坏情况输入生成

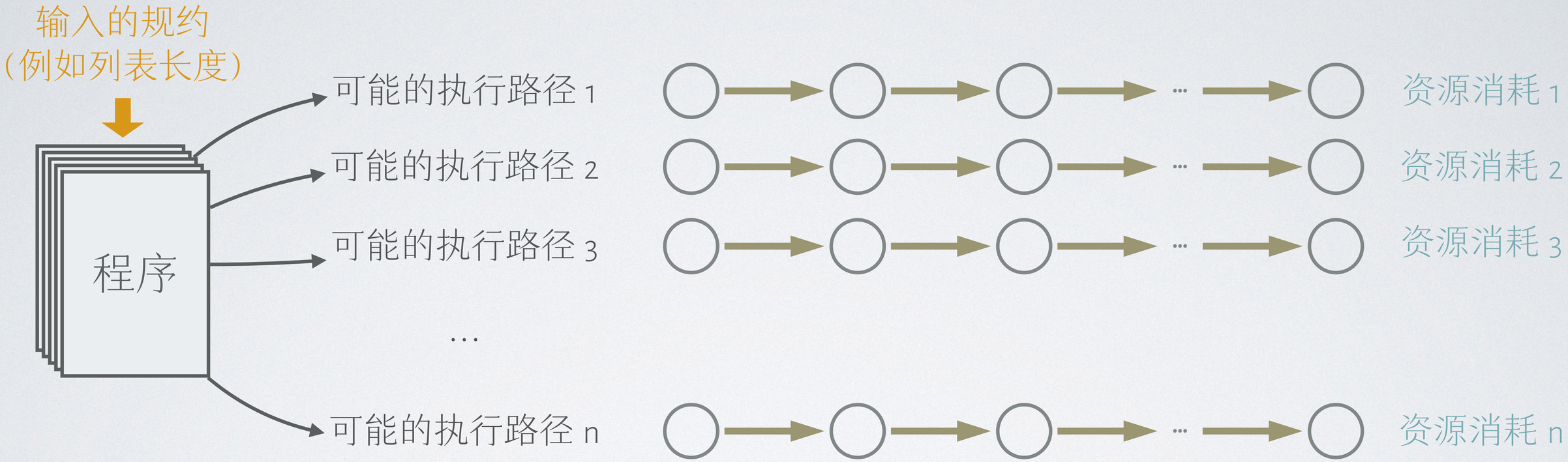


基于符号执行的最坏情况输入生成



- 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输入生成



- 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？

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

资源制导的符号执行

$\langle L^1(\text{int}), 0\rangle \to \langle L^0(\text{int} \times \text{int}), 0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

Cost=2

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost=2

$\Phi' = |xs'| = 2$

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost=2

$\Phi' = |xs'| = 2$

资源制导的符号执行

$\langle L^1(\text{int}),0\rangle \to \langle L^0(\text{int}\times\text{int}),0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost=2

$\Phi' = |xs'| = 2$

势能浪费！

资源制导的符号执行

$\langle L^1(\text{int}), 0\rangle \to \langle L^0(\text{int} \times \text{int}), 0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost=2

$\Phi' = |xs'| = 2$

势能浪费！

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

资源制导的符号执行

$\langle L^1(\text{int}), 0\rangle \to \langle L^0(\text{int} \times \text{int}), 0\rangle$

```
let rec lpairs l =
  match l with
  | [] -> []
  | x1::xs ->
    match xs with
    | [] -> []
    | x2::xs' ->
      if x1 < x2 then
        let () = tick(2) in
        (x1,x2)::(lpairs xs')
      else
        lpairs xs'
```

$\ell \mapsto [\text{int}^1, \text{int}^2, \text{int}^3, \text{int}^4]$

$\Phi = |xs'| + 2 = 4$

$x_1 \mapsto \text{int}^1, x_2 \mapsto \text{int}^2,$

$xs' \mapsto [\text{int}^3, \text{int}^4]$

Cost=2

$\Phi' = |xs'| = 2$

势能浪费！

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

通过这个信息减少搜索量！

从分析到验证：量化霍尔逻辑

Hoare Logic

$\{\Gamma\}\ S\ \{\Gamma'\}$

从分析到验证：量化霍尔逻辑

Hoare Logic

$\{\Gamma\}\ S\ \{\Gamma'\}$

$\Gamma, \Gamma' : State \to bool$

从分析到验证：量化霍尔逻辑

Hoare Logic

$$\{\Gamma\}\ S\ \{\Gamma'\}$$

$$\Gamma, \Gamma' : State \to bool$$

如果初始状态 σ 满足 $\Gamma(\sigma)$ 为真
且运行程序 S 到达终止状态 σ'
那么 $\Gamma'(\sigma')$ 也为真

从分析到验证：量化霍尔逻辑

Hoare Logic	Quantitative Hoare Logic
$\{\Gamma\}\ S\ \{\Gamma'\}$	$\{\Phi\}\ S\ \{\Phi'\}$
$\Gamma, \Gamma' : State \to bool$	$\Phi, \Phi' : State \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$
如果初始状态 σ 满足 $\Gamma(\sigma)$ 为真 且运行程序 S 到达终止状态 σ' 那么 $\Gamma'(\sigma')$ 也为真	

从分析到验证：量化霍尔逻辑

Hoare Logic	Quantitative Hoare Logic
$\{\Gamma\}\ S\ \{\Gamma'\}$	$\{\Phi\}\ S\ \{\Phi'\}$
$\Gamma, \Gamma' : State \to bool$	$\Phi, \Phi' : State \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$
如果初始状态 σ 满足 $\Gamma(\sigma)$ 为真 且运行程序 S 到达终止状态 σ' 那么 $\Gamma'(\sigma')$ 也为真	如果初始有至少 $\Phi(\sigma)$ 单位的势能 且运行程序 S 从 σ 到达 σ' 那么终止有至少 $\Phi'(\sigma')$ 单位的势能

从分析到验证：量化霍尔逻辑

Hoare Logic	Quantitative Hoare Logic
$\{\Gamma\}\ S\ \{\Gamma'\}$	$\{\Phi\}\ S\ \{\Phi'\}$
$\Gamma, \Gamma' : State \to bool$	$\Phi, \Phi' : State \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$
$false$	∞
如果初始状态 σ 满足 $\Gamma(\sigma)$ 为真 且运行程序 S 到达终止状态 σ' 那么 $\Gamma'(\sigma')$ 也为真	如果初始有至少 $\Phi(\sigma)$ 单位的势能 且运行程序 S 从 σ 到达 σ' 那么终止有至少 $\Phi'(\sigma')$ 单位的势能

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**
- **Incorrectness Logic!** [1]

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**
- **Incorrectness Logic!** [1]

$$[p]\ C\ [\varepsilon: q] \quad \textit{iff} \quad \forall\, s \in q.\ \exists\, s' \in p.\ (s', s) \in [C]\varepsilon$$

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**
- **Incorrectness Logic!** [1]

$$[p]\ C\ [\varepsilon: q] \quad \textit{iff} \quad \forall\, s \in q.\ \exists\, s' \in p.\ (s', s) \in [C]\varepsilon$$

- 进行中工作：量化下近似逻辑

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**
- **Incorrectness Logic!** [1]

$$[p]\ C\ [\varepsilon: q] \quad \textit{iff} \quad \forall s \in q.\ \exists s' \in p.\ (s', s) \in [C]\varepsilon$$

- 进行中工作：量化下近似逻辑

$$\vDash_B [P]C[Q] \quad P, Q: \Sigma \to \mathbb{R}^{\pm\infty}$$
$$\forall \sigma, p \text{ s.t. } p \leq P(\sigma).\ \exists \tau, q \text{ s.t. } (C, \sigma, p) \to (\text{skip}, \tau, q).\ q \leq Q(\tau)$$

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

工作 4：最坏情况资源消耗下近似验证

Z. Jin and D. Wang. Underapproximation Logic for Worst-Case Resource Analysis. *Working Paper*.

- 观察：量化霍尔逻辑对资源消耗进行上近似验证，而输入生成进行单点下近似
- **问题：如何对资源消耗进行下近似验证？**
- **Incorrectness Logic!** [1]

$[p]\ C\ [\varepsilon: q] \quad \textit{iff} \quad \forall s \in q.\ \exists s' \in p.\ (s', s) \in [C]\varepsilon$

- 进行中工作：量化下近似逻辑

$$\vDash_B [P]C[Q] \quad P, Q : \Sigma \to \mathbb{R}^{\pm\infty}$$
$$\forall \sigma, p \text{ s.t. } p \le P(\sigma).\ \exists \tau, q \text{ s.t. } (C, \sigma, p) \to (\text{skip}, \tau, q).\ q \le Q(\tau)$$

$$\vDash^{\diamond}_B [P]C[Q] \quad P, Q : \Sigma \to \mathbb{R}^{\pm\infty}$$
$$\forall \sigma, p \text{ s.t. } p \le P(\sigma).\ \exists \tau, q \text{ s.t. } (C, \sigma, p) \to (\text{skip}, \tau, q).\ q \le Q(\tau)$$

求值过程中势能降到过 0

[1] P. W. O'Hearn. 2020. Incorrectness Logic. In *POPL'20*.

程序的资源分析与验证

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

程序的资源分析与验证

- ☑ 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 的流行说明了在类型系统中追踪**更多的性质**对编程语言是可以有益的

程序的资源分析与验证

- ☑ 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 的流行说明了在类型系统中追踪**更多的性质**对编程语言是可以有益的
- 如何**从资源保障的需求出发**，改进 Rust 或者设计新型资源敏感编程语言？