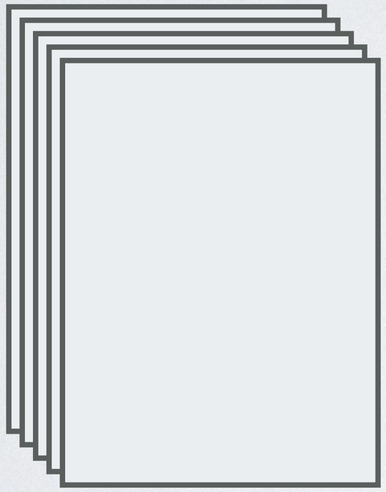
Type-Guided Worst-Case Input Generation

Di Wang

About Me

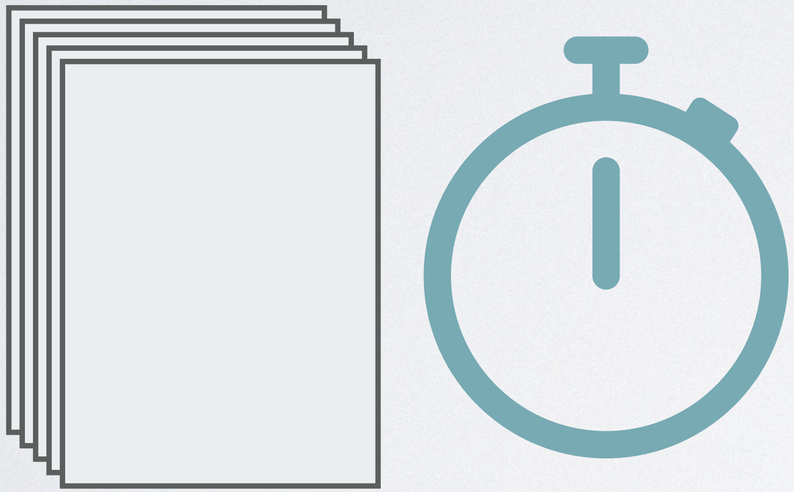
- I am a fourth-year doctoral student in Computer Science
- I am interested in programming languages and software engineering
- My focuses are **probabilistic programming** and **static resource analysis**

Resource Analysis



Programs

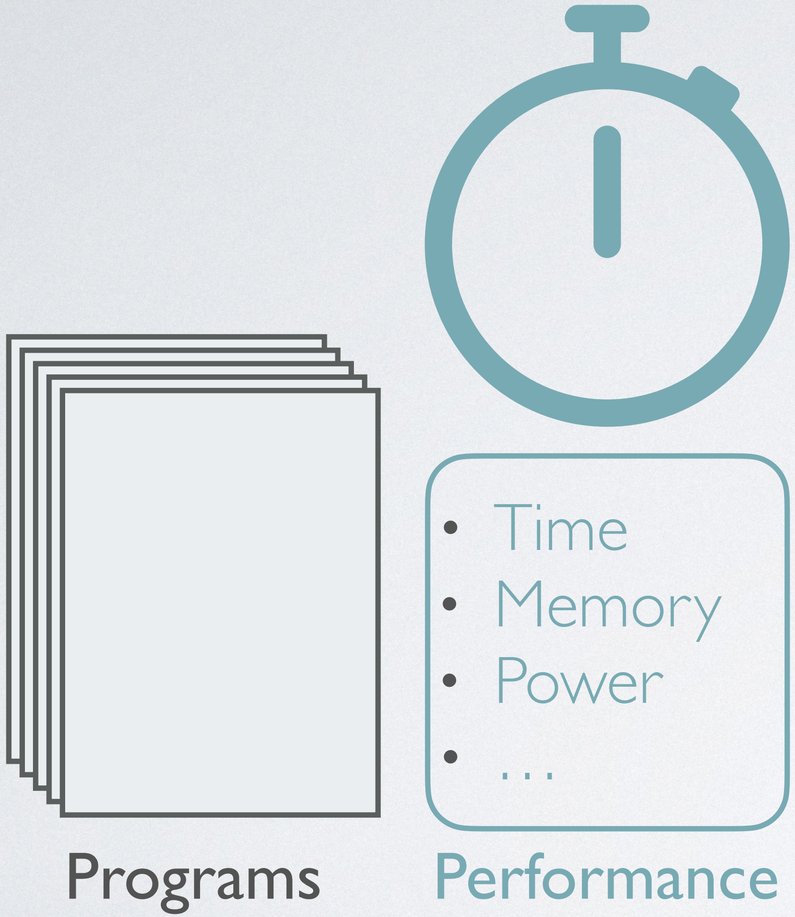
Resource Analysis



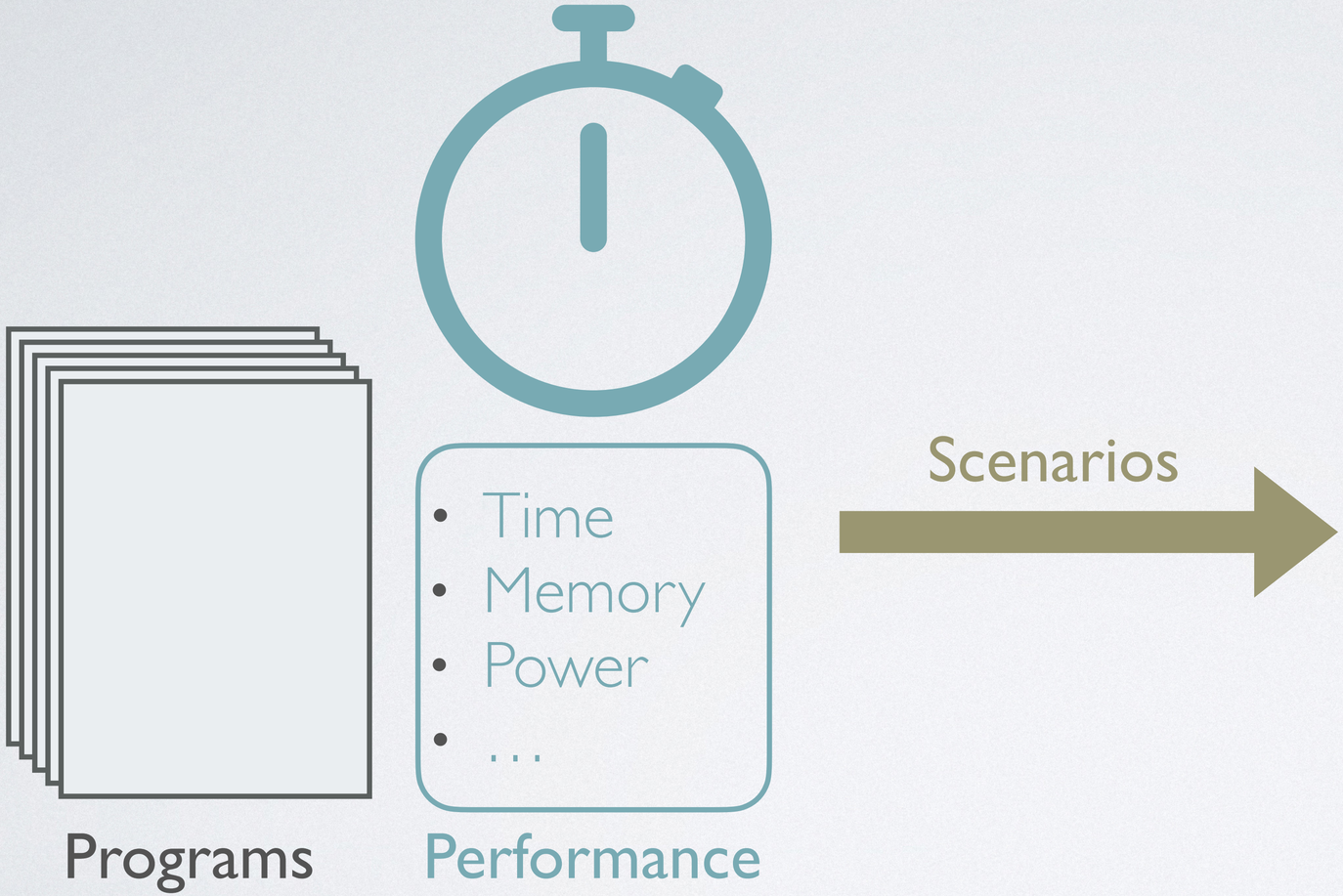
Programs

Performance

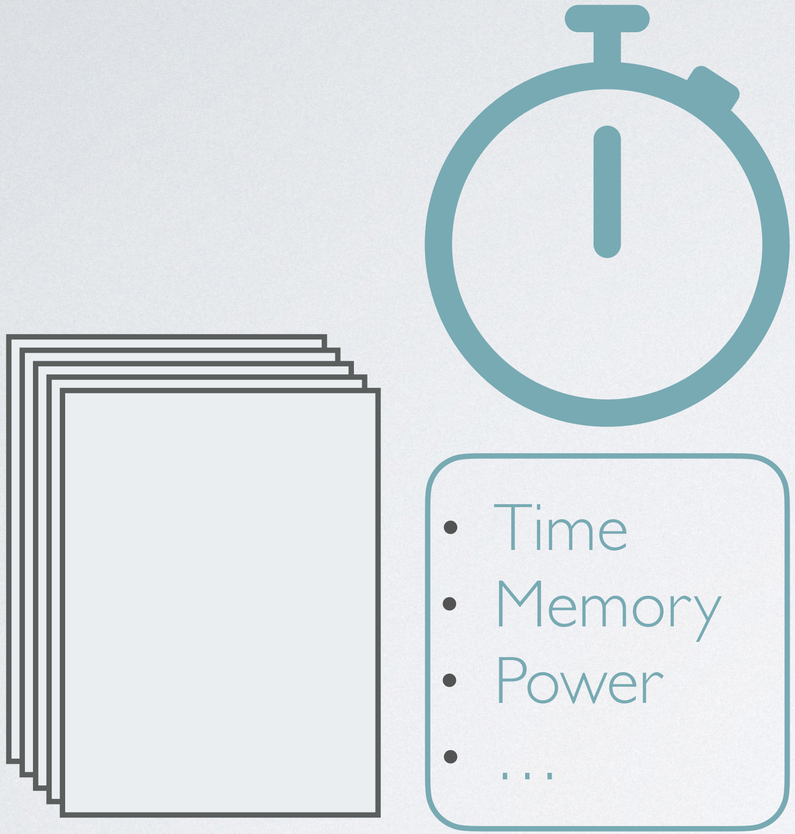
Resource Analysis



Resource Analysis



Resource Analysis



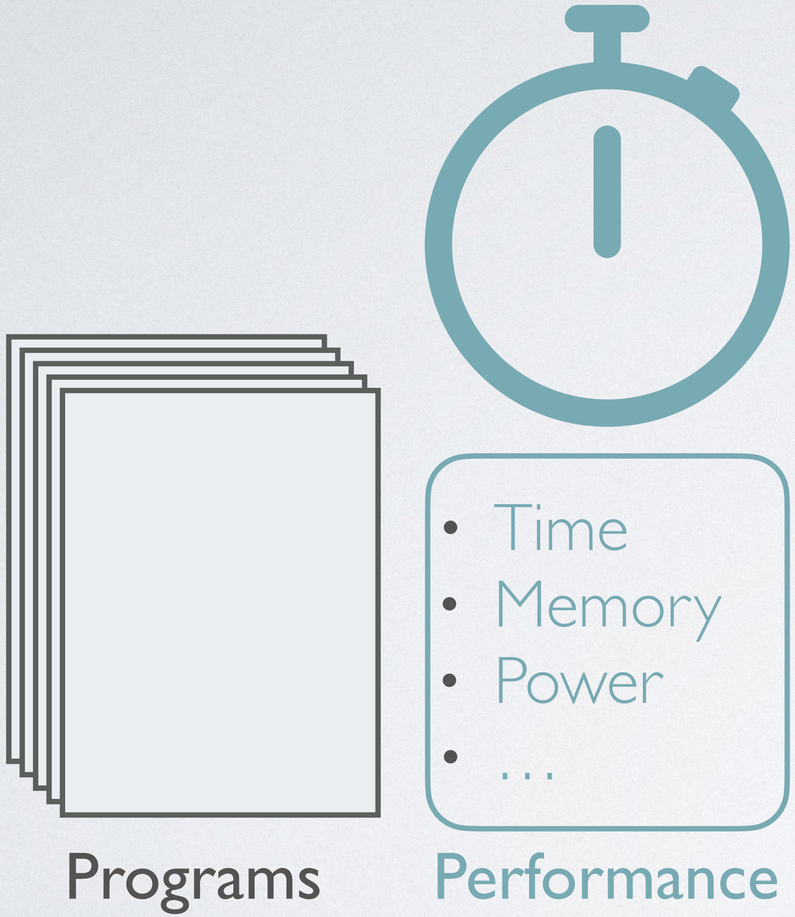
Programs

Performance

Scenarios

- Identifying bottlenecks

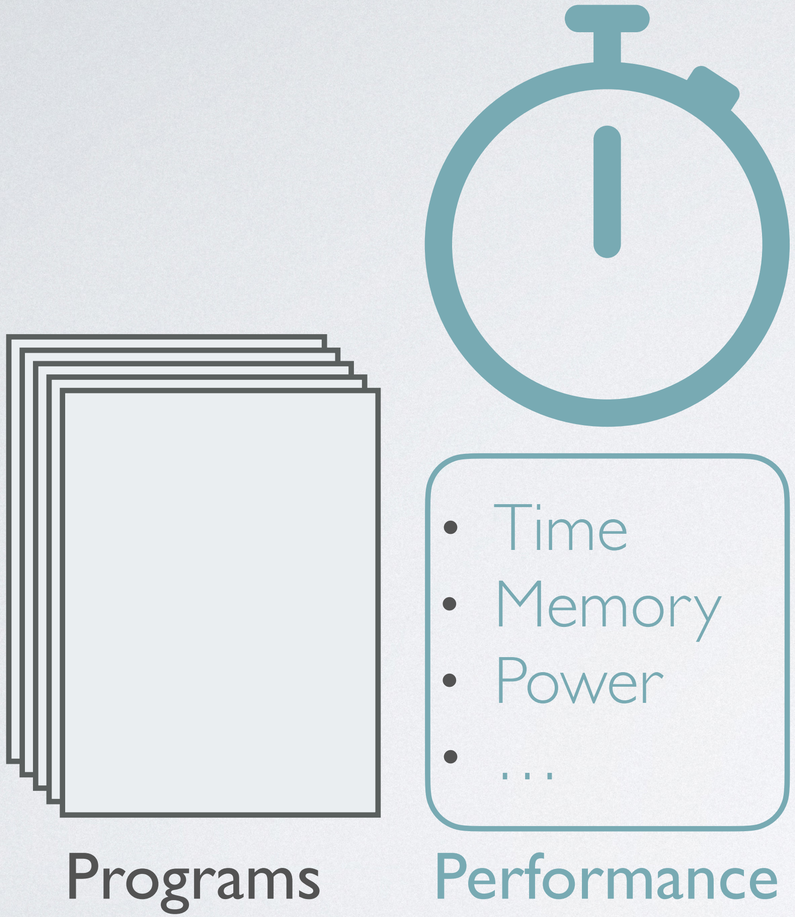
Resource Analysis



Scenarios

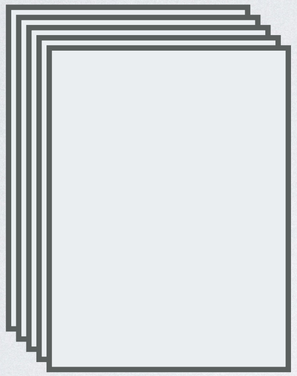
- Identifying bottlenecks
- Gas usage in blockchains

Resource Analysis



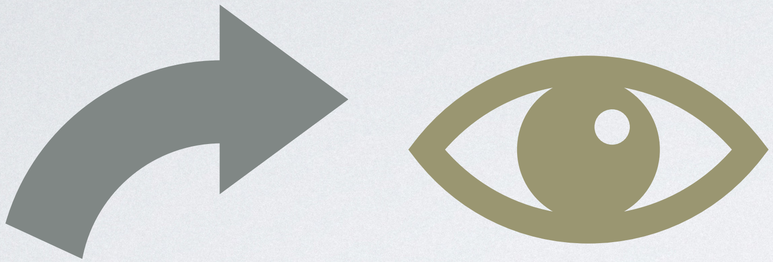
- Identifying bottlenecks
- Gas usage in blockchains
- Carbon footprint

Static Resource Analysis

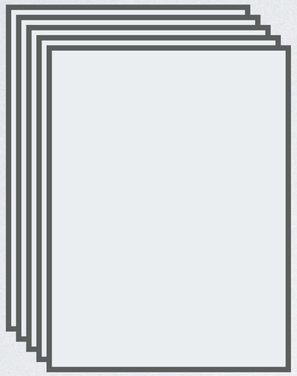


New Commits

Static Resource Analysis

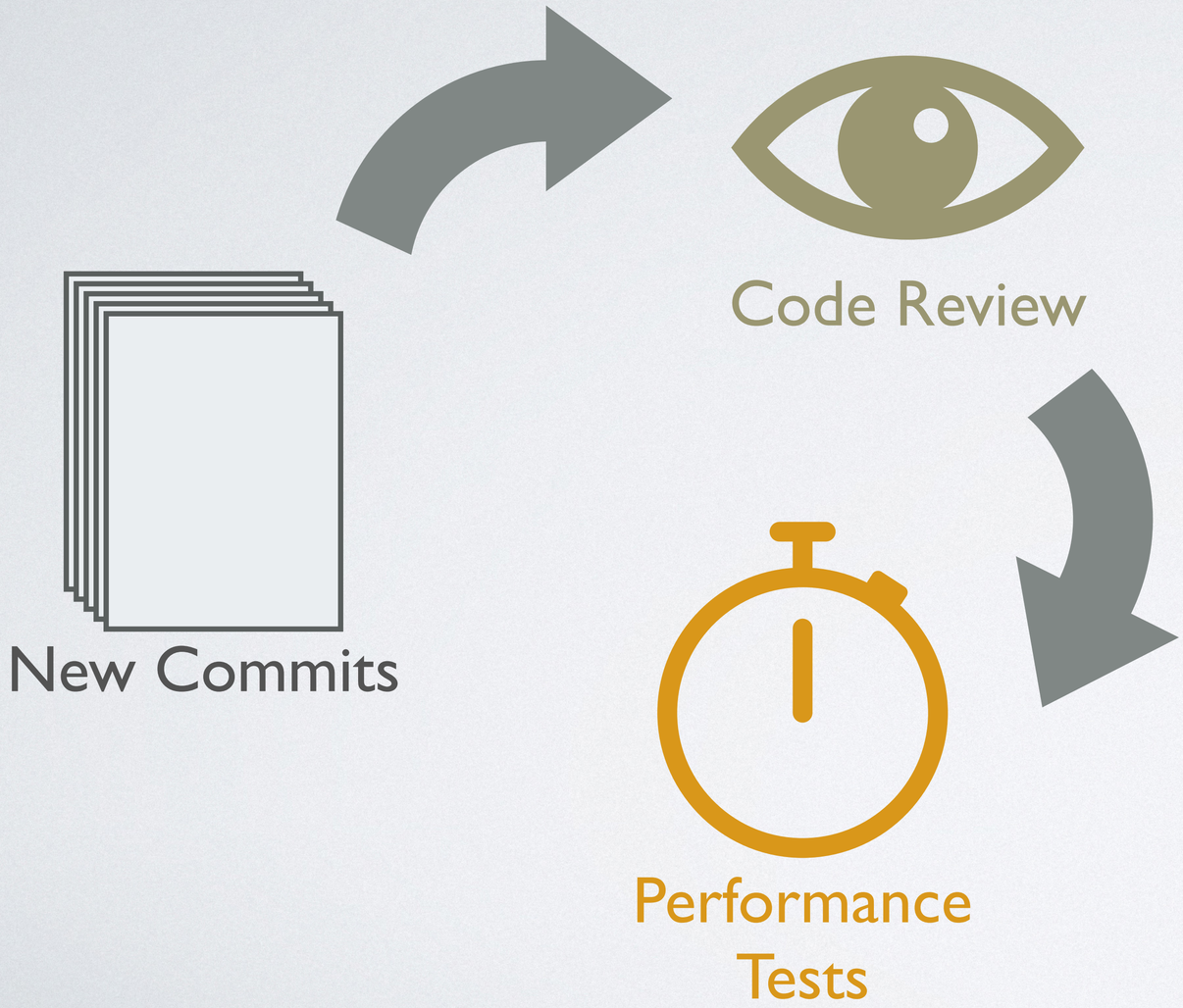


Code Review

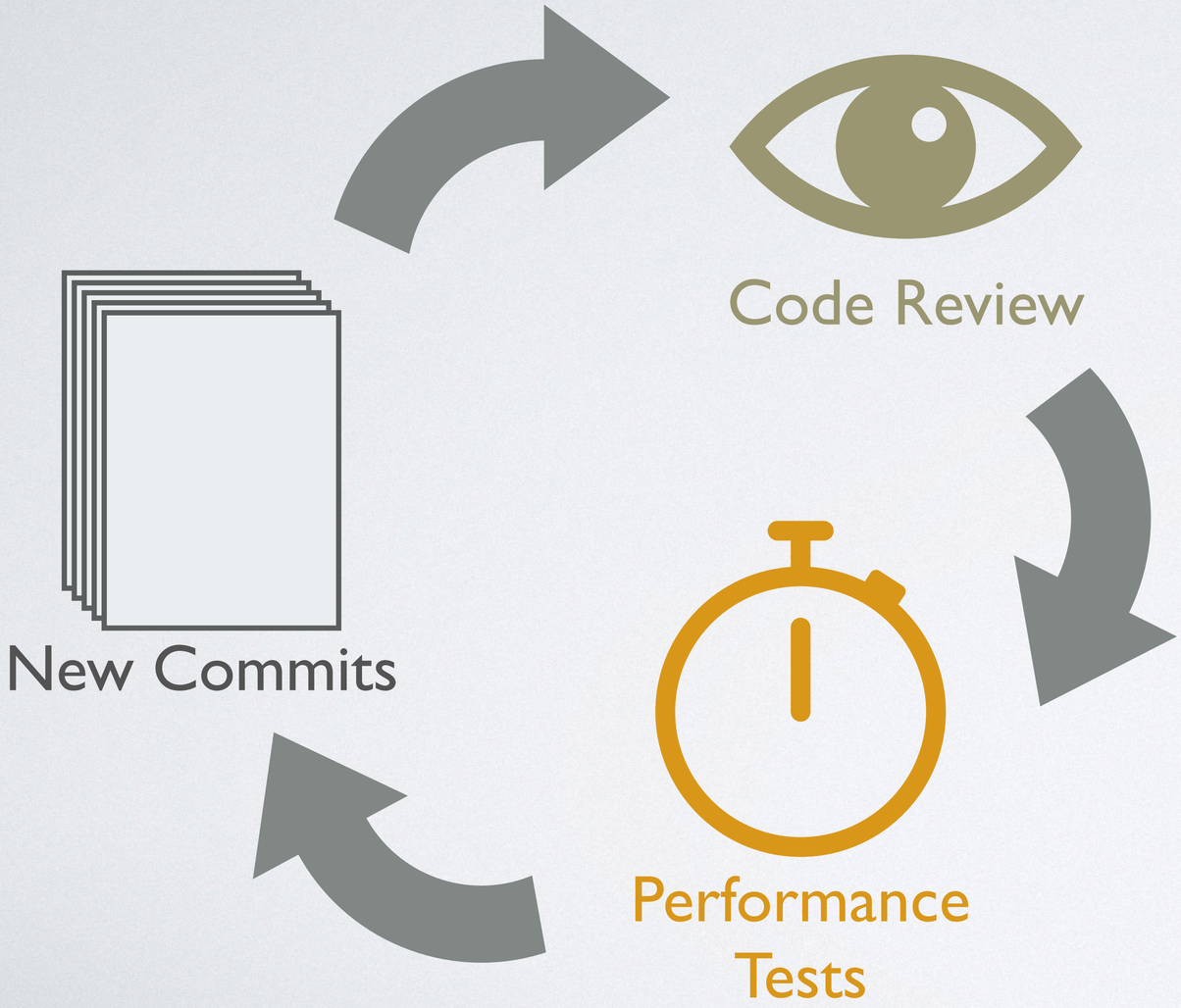


New Commits

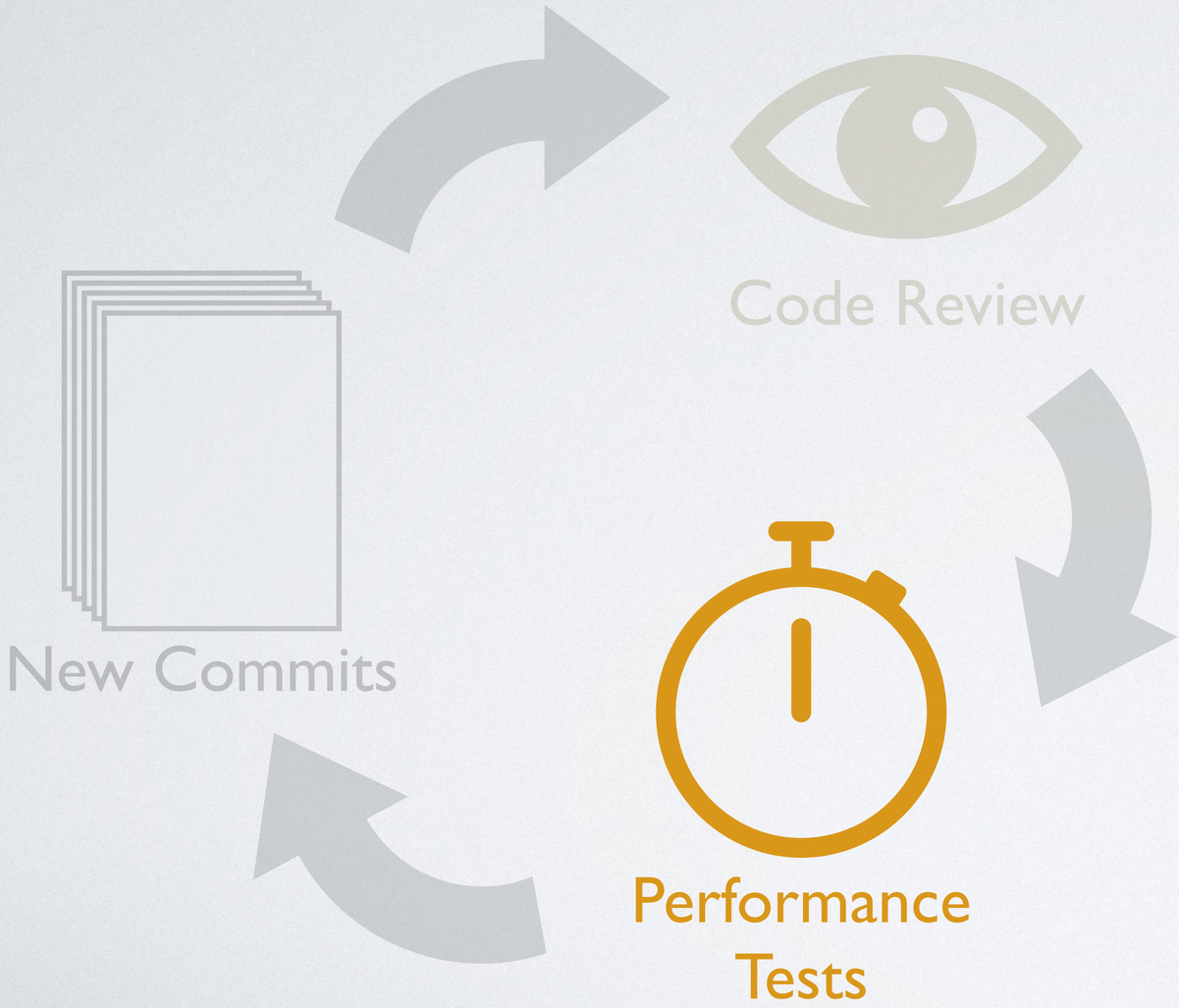
Static Resource Analysis



Static Resource Analysis



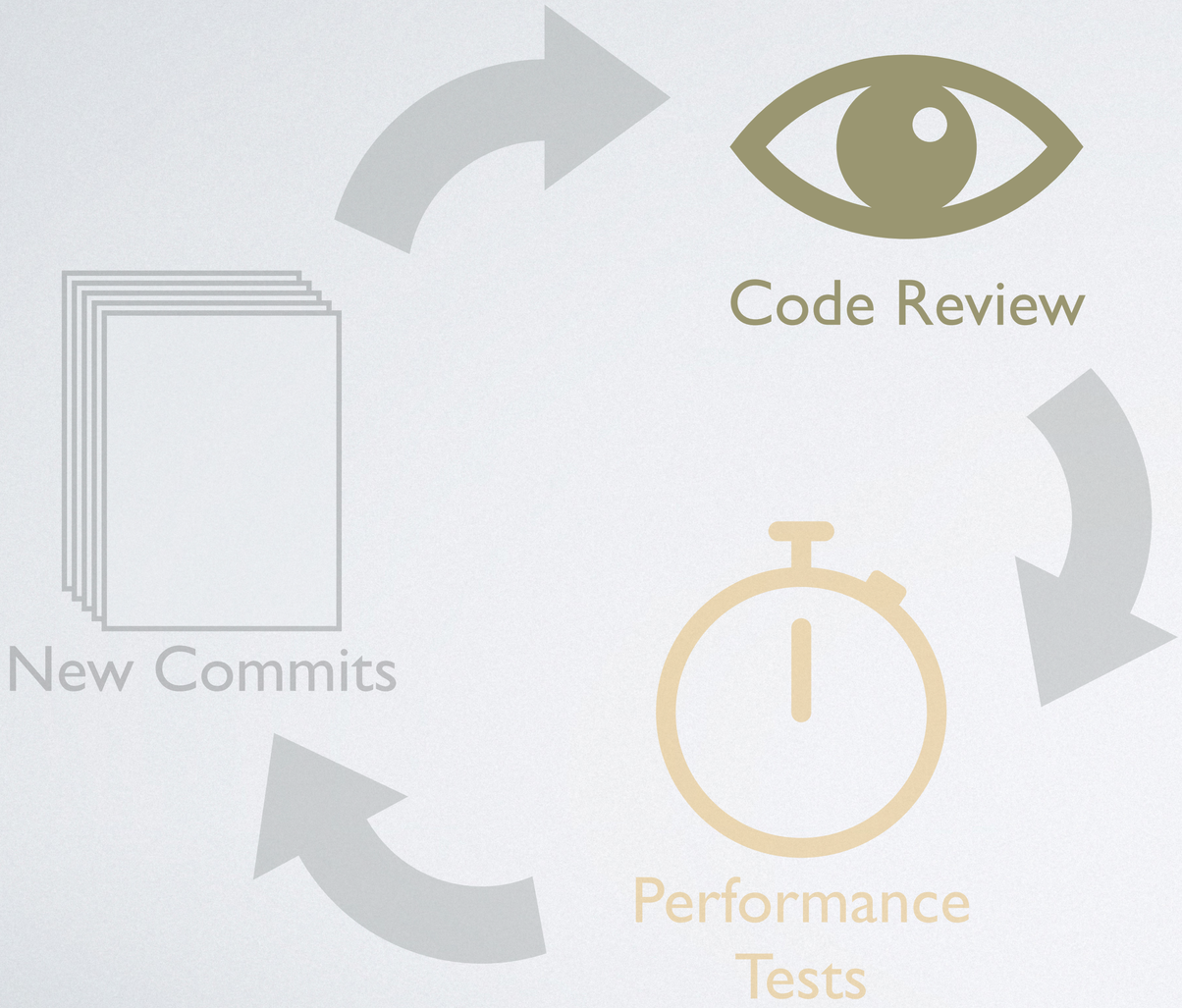
Static Resource Analysis



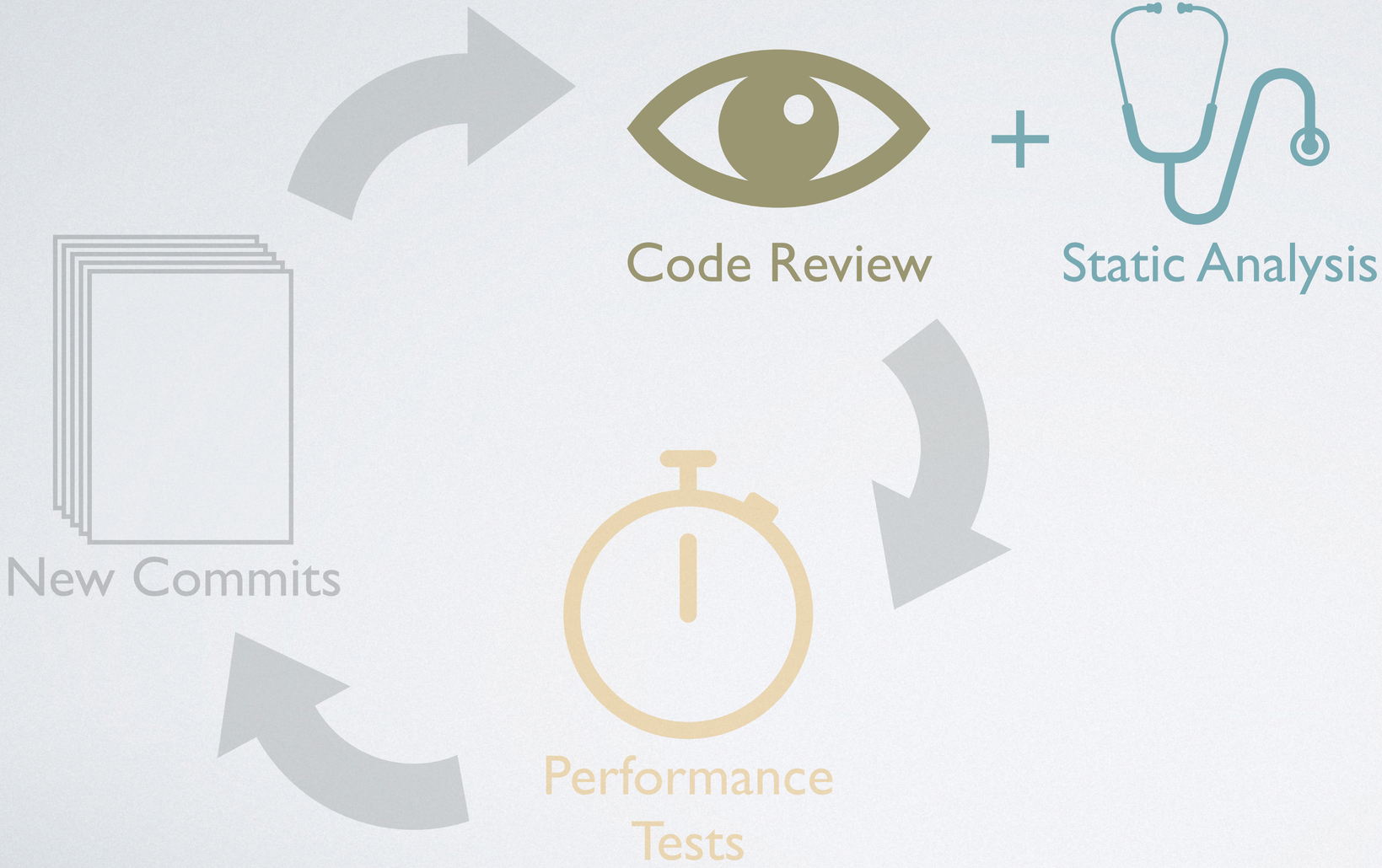
Possible drawbacks:

- Incomplete test coverage
- Time-consuming

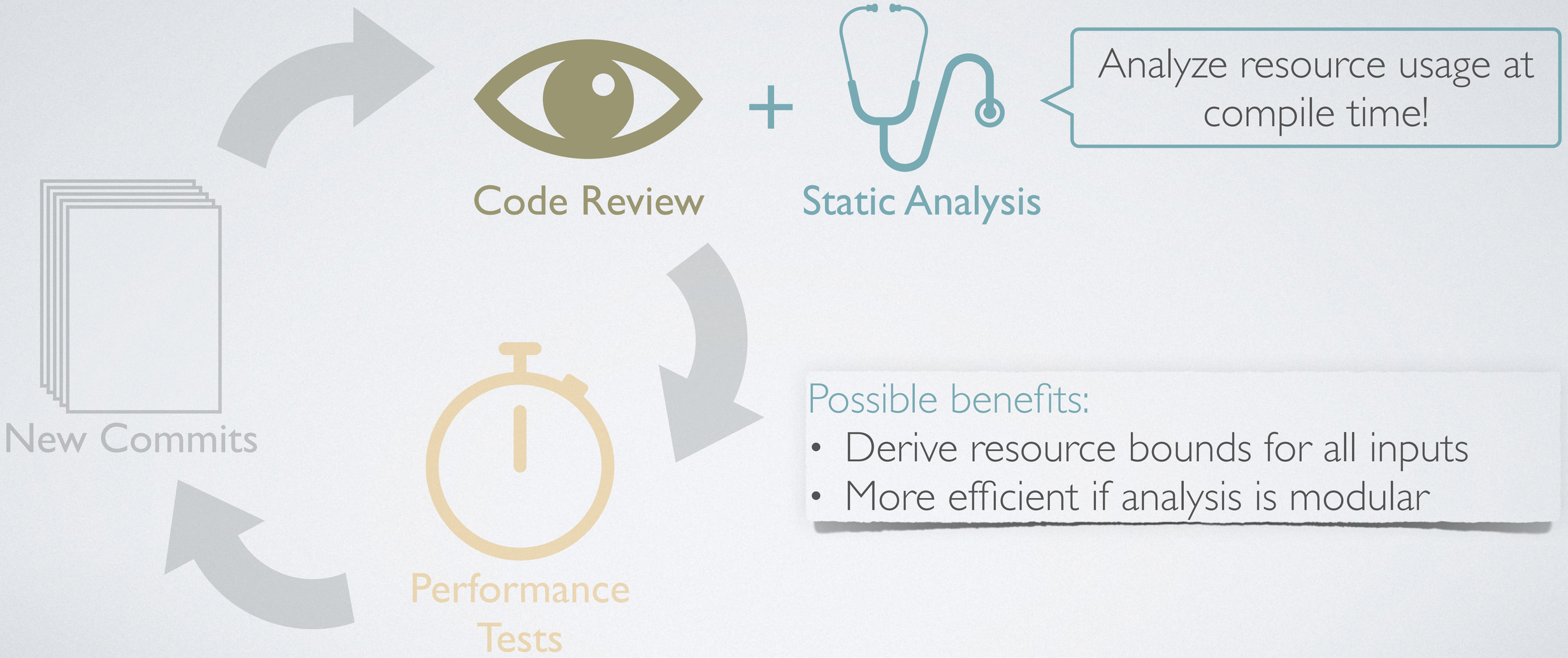
Static Resource Analysis



Static Resource Analysis



Static Resource Analysis



Possible benefits:

- Derive resource bounds for all inputs
- More efficient if analysis is modular

STATIC RESOURCE ANALYSIS IN INFER

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

STATIC RESOURCE ANALYSIS IN INFER

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

STATIC RESOURCE ANALYSIS IN INFER

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

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

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

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

```
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++) {
    foo(i); // newly added function call
  }
}
```

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

```
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++) {
    foo(i); // newly added function call
  }
}
```

$O(|list|^2)$

The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Static Resource Analysis in Infer

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

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

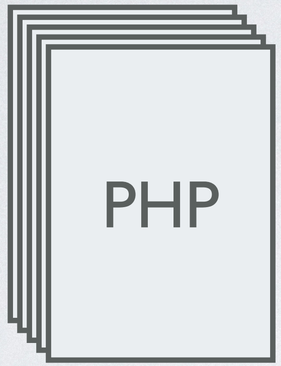
$8|list| + 16 = O(|list|)$

Complexity increase!

$O(|list|^2)$

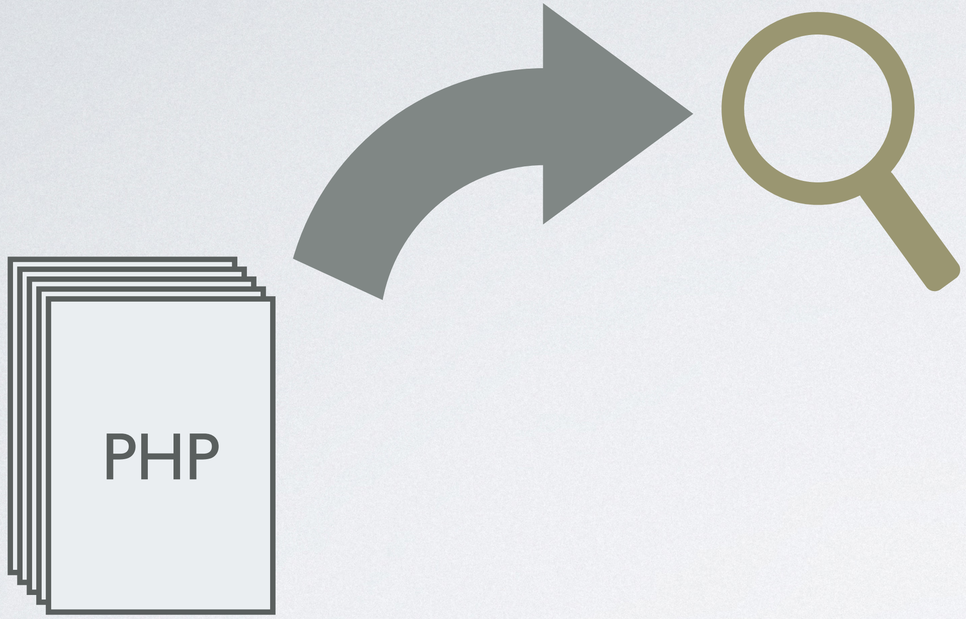
The example comes from Infer's documentation. Available on: https://fbinfer.com/docs/next/checker-cost/.

Worst-Case Analysis



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

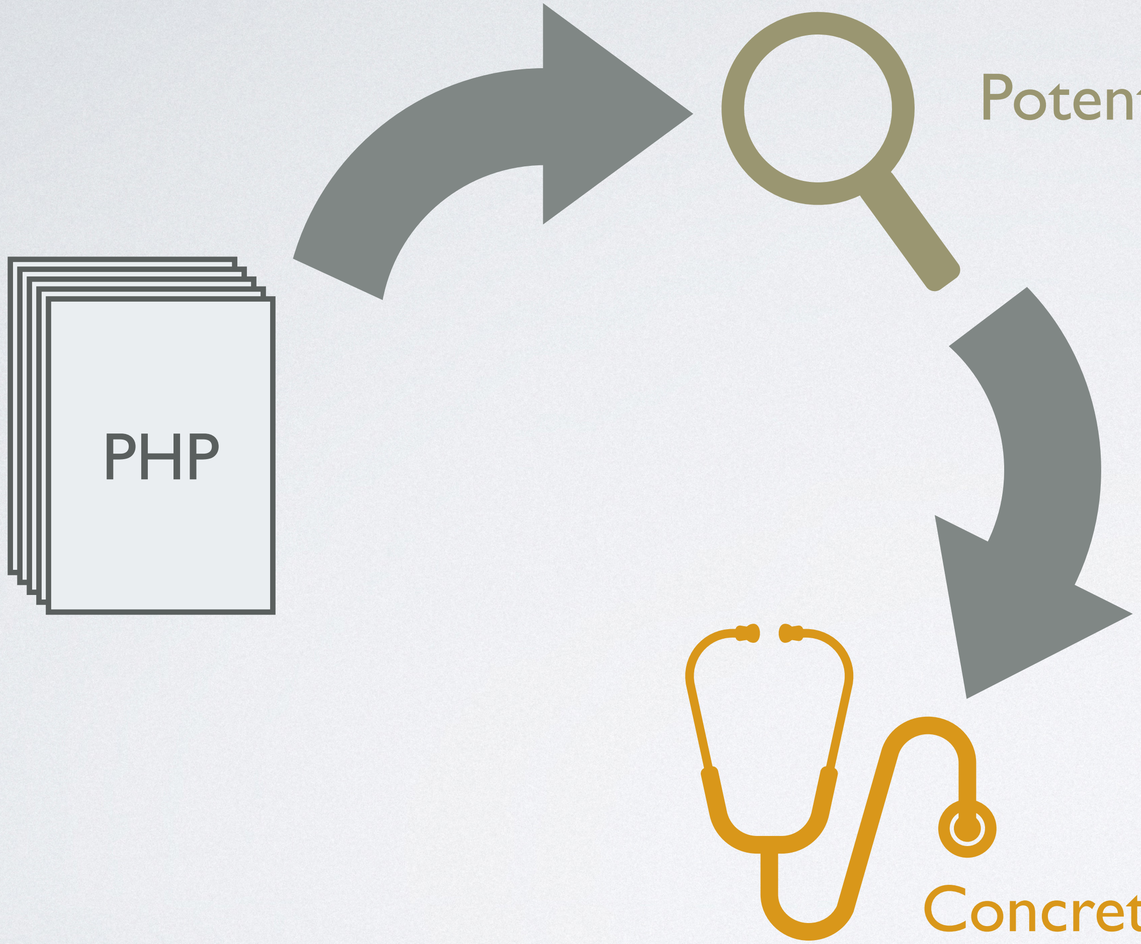
Worst-Case Analysis



Potential Denial-of-Service attack[1]

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

Worst-Case Analysis



Potential Denial-of-Service attack[1]

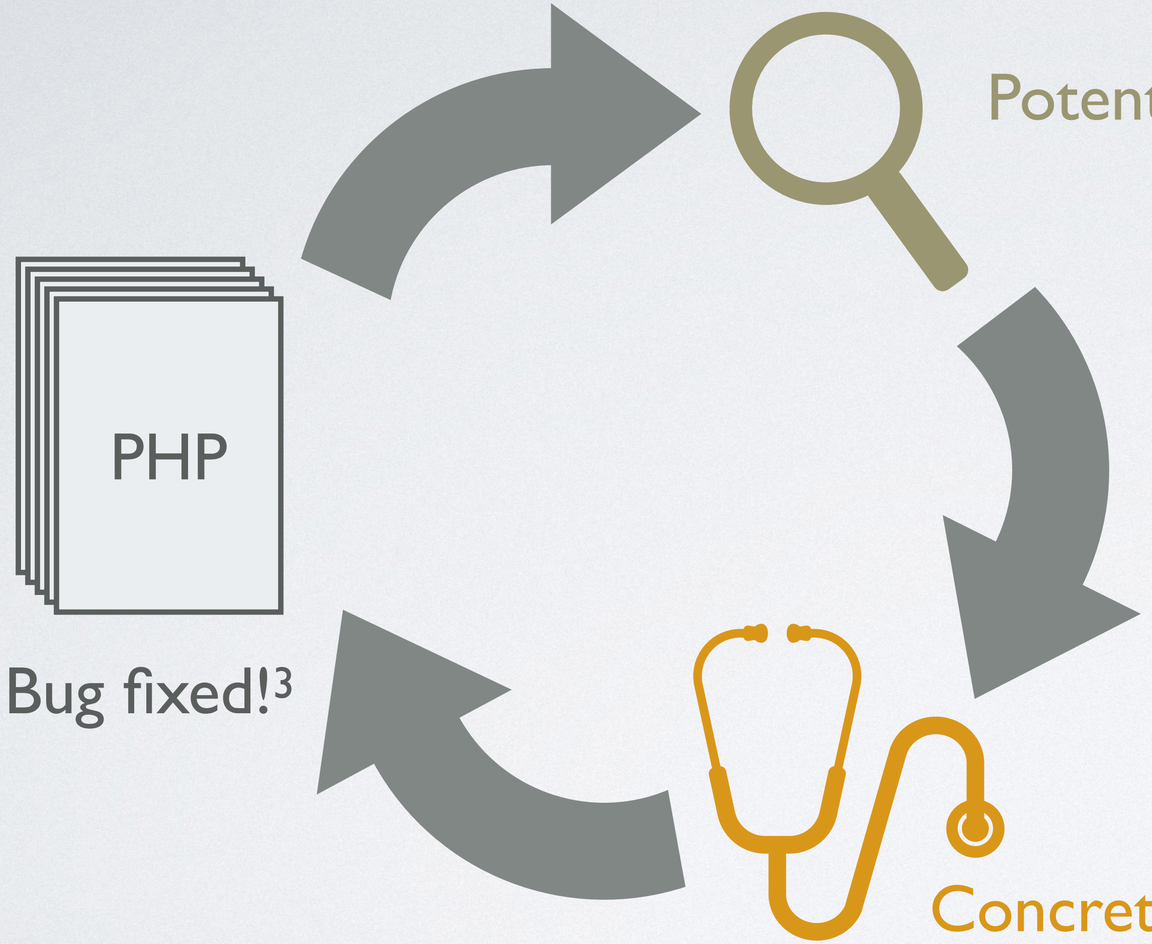
Concrete exploits (by hash collisions)[2]

[1] CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.

[2] PHP 5.3.8 - Hashtables Denial of Service. Available on https://www.exploit-db.com/exploits/18296/.

[3] PHP: PHP 5 ChangeLog. Available on http://www.php.net/ChangeLog-5.php#5.3.9.

Worst-Case Analysis



Potential Denial-of-Service attack[1]

Concrete exploits (by hash collisions)[2]

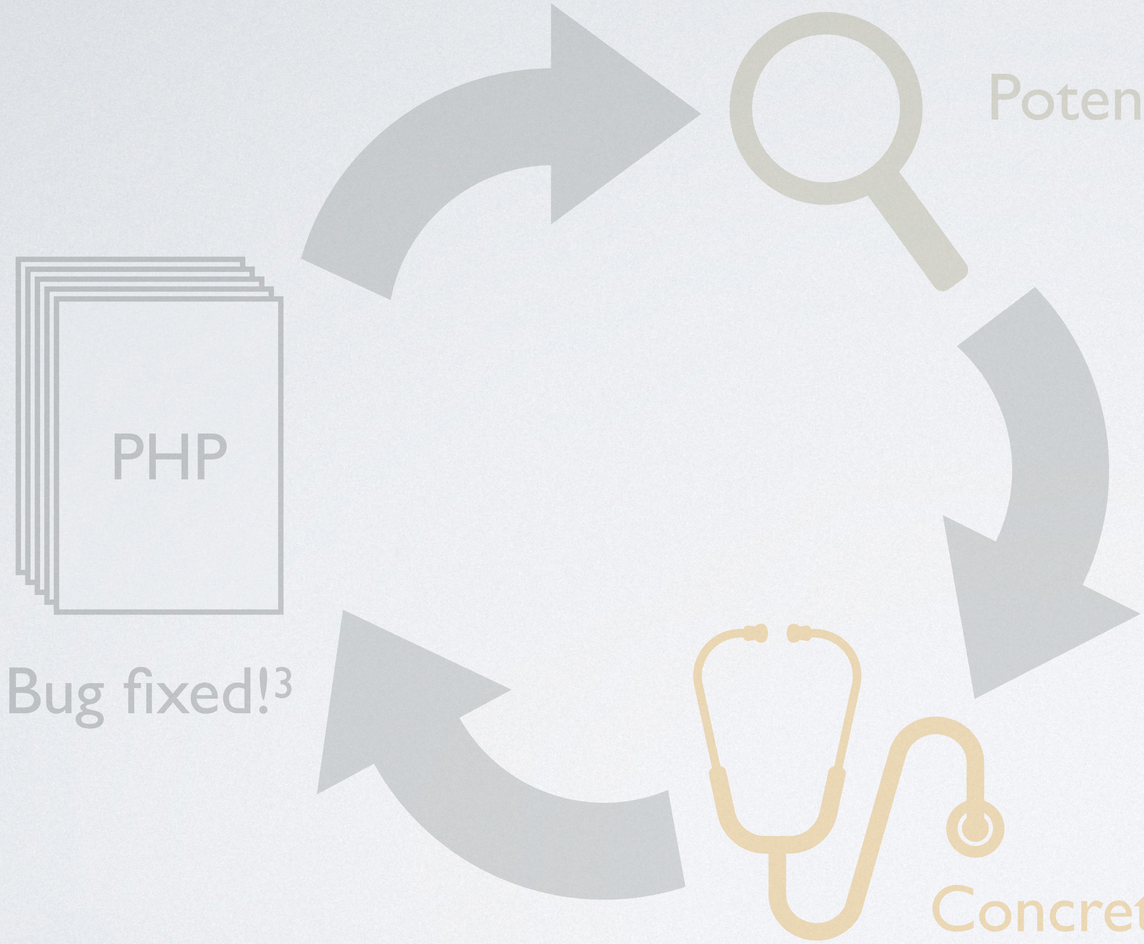
Bug fixed![3]

[1] CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.

[2] PHP 5.3.8 - Hashtables Denial of Service. Available on https://www.exploit-db.com/exploits/18296/.

[3] PHP: PHP 5 ChangeLog. Available on http://www.php.net/ChangeLog-5.php#5.3.9.

Worst-Case Analysis



Potential Denial-of-Service attack[1]

Concrete exploits (by hash collisions)[2]

Bug fixed![3]

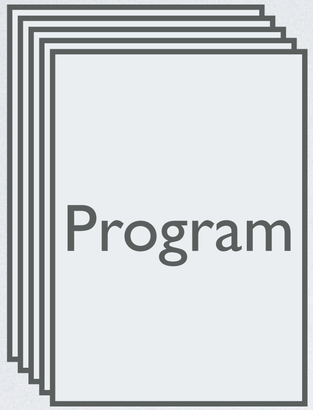
Worst-case inputs are instrumental in understanding and fixing performance bugs!

[1] CVE - CVE-2011-4885. Available on: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4885.

[2] PHP 5.3.8 - Hashtables Denial of Service. Available on https://www.exploit-db.com/exploits/18296/.

[3] PHP: PHP 5 ChangeLog. Available on http://www.php.net/ChangeLog-5.php#5.3.9.

Worst-Case Analysis

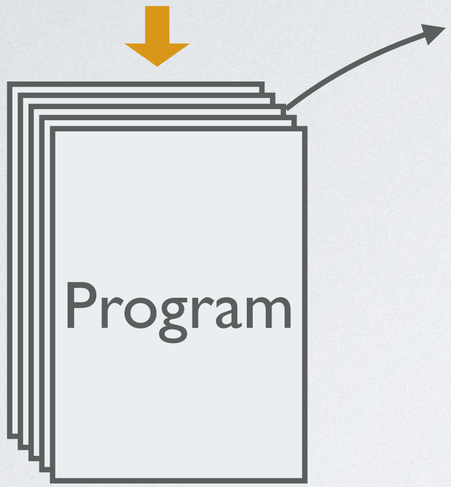


Worst-Case Analysis

Input specification
(e.g., a list of length 4)

Worst-Case Analysis

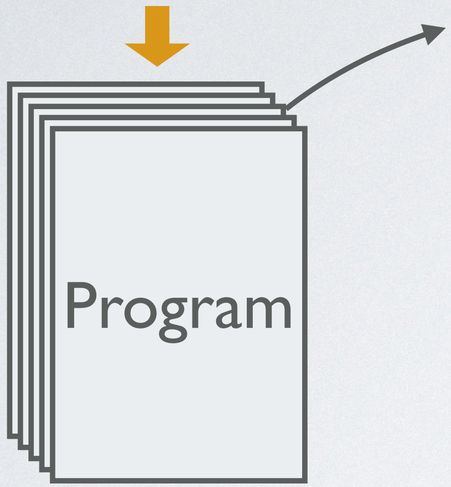
Input specification
(e.g., a list of length 4)



Possible execution path 1

Worst-Case Analysis

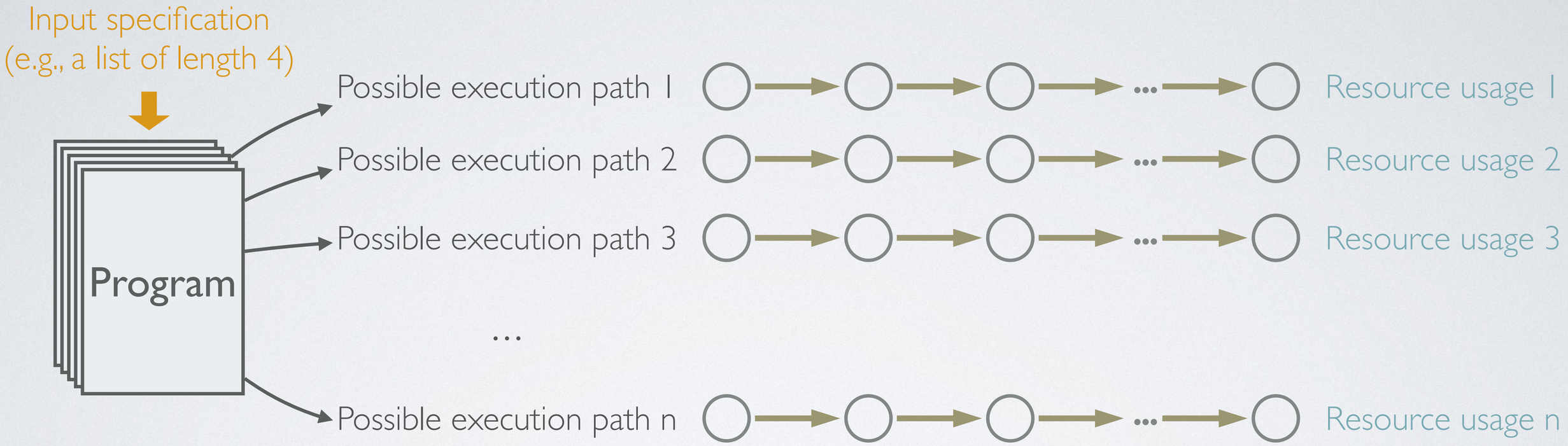
Input specification
(e.g., a list of length 4)



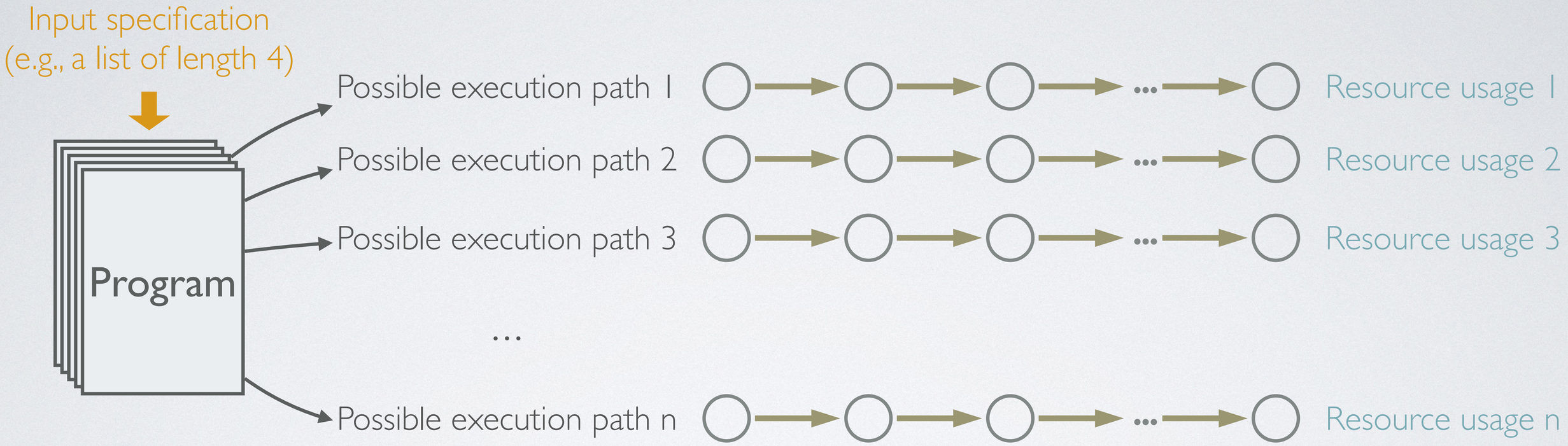
Possible execution path 1

Resource usage 1

Worst-Case Analysis

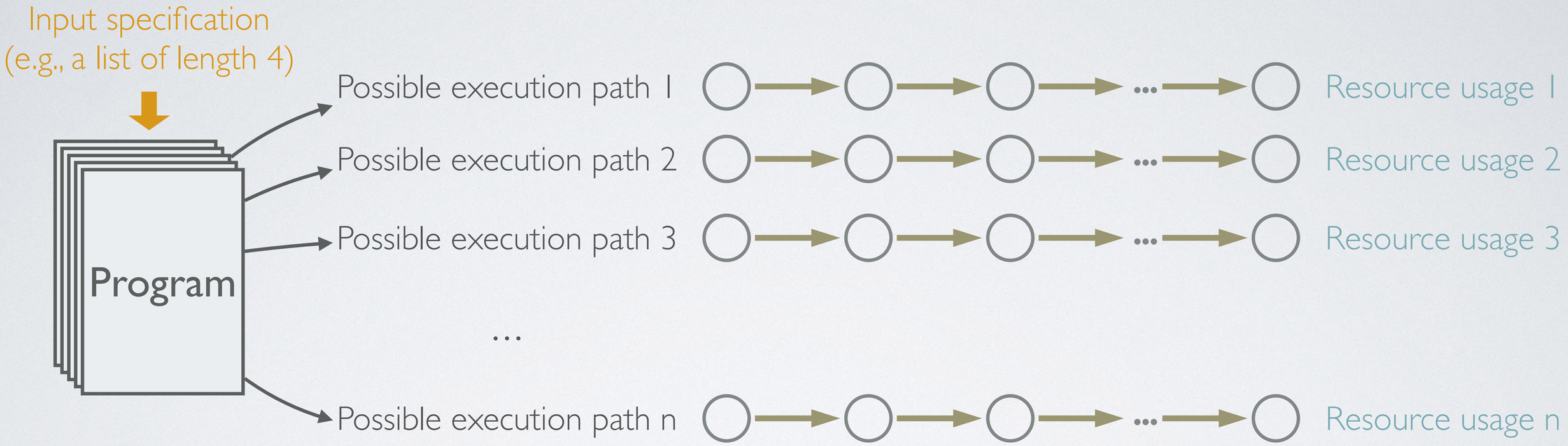


Worst-Case Analysis



- **Search** for a program execution path with the **maximal resource usage**

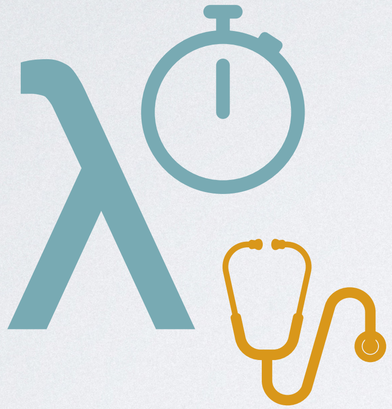
Worst-Case Analysis



- Search for a program execution path with the maximal resource usage
- Can be very inefficient

Type-Guided Worst-Case Input Generation

Di Wang and Jan Hoffmann. 2019. Published in *POPL'19*.



Type-Guided Worst-Case Input Generation

Di Wang and Jan Hoffmann. 2019. Published in *POPL'19*.

Static Resource Analysis (Type-Based)

Type-Guided Worst-Case Input Generation

Di Wang and Jan Hoffmann. 2019. Published in *POPL'19*.

Static Resource Analysis (Type-Based)

Worst-Case Execution-Path Search

Type-Guided Worst-Case Input Generation

Di Wang and Jan Hoffmann. 2019. Published in *POPL'19*.

Type-Guided Worst-Case Input Generation

Di Wang and Jan Hoffmann. 2019. Published in *POPL'19*.

The first provably correct worst-case input generation algorithm based on static resource analysis

Overview

- [x] Motivation
- [ ] Type-Based Resource Analysis
- [ ] Type-Guided Worst-Case Input Generation
- [ ] Evaluation

Type-Based Resource Analysis

OCaml

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

RaML

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

Type-Based Resource Analysis

OCaml

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

append : $\langle L^9(\text{int}) \times L^0(\text{int}), 3\rangle \to \langle L^0(\text{int}), 0\rangle$

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

Type-Based Resource Analysis

OCaml

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

append : A Resource-Annotated Type

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

Type-Based Resource Analysis

OCaml

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

append : A Resource-Annotated Type

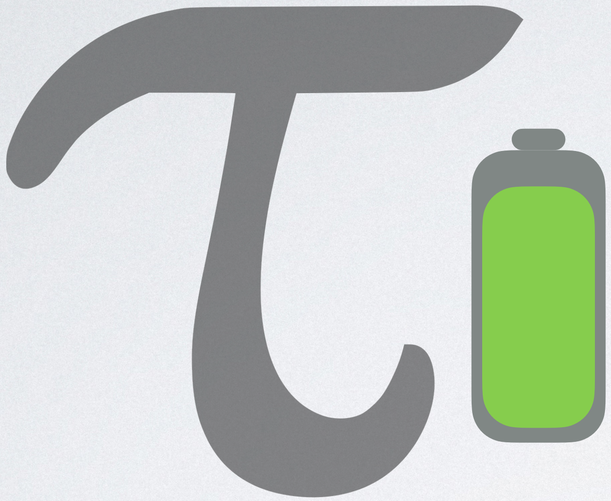
The simplified upper bound on evaluation steps:

$$9|\ell_1| + 3 = O(|\ell_1|)$$

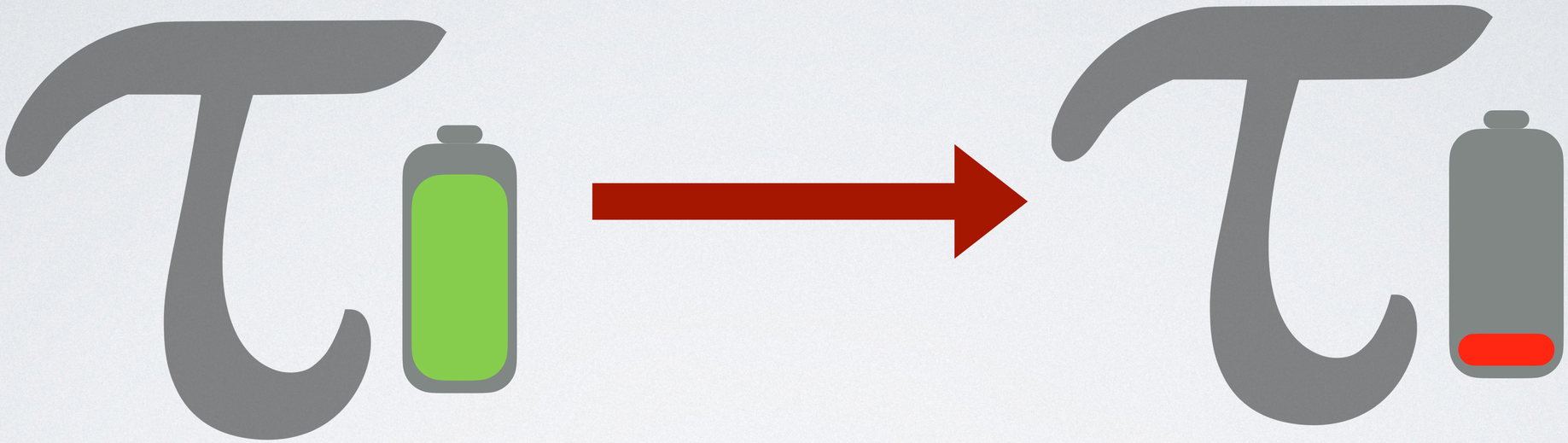
[RaML] J. Hoffmann, A. Das, and S.-C. Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL'17*.

Type-Based Resource Analysis

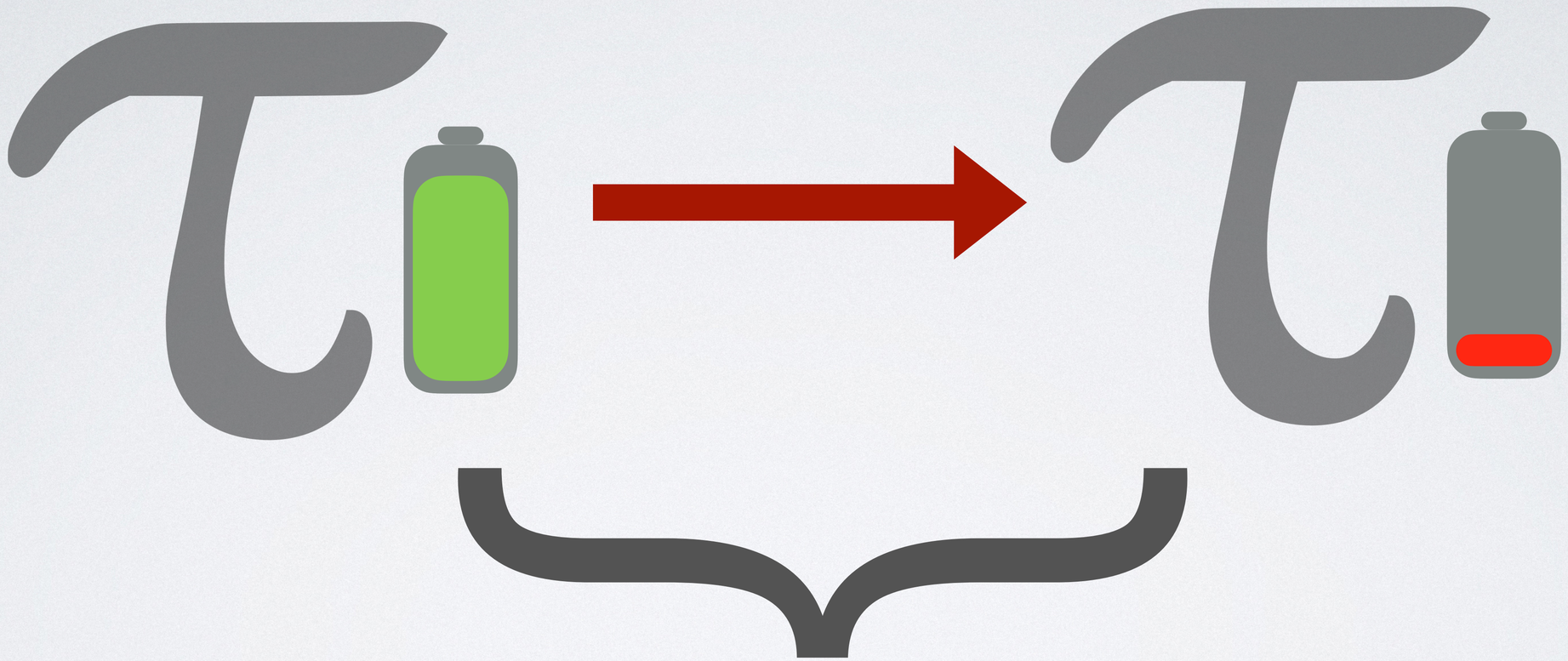
Type-Based Resource Analysis



Type-Based Resource Analysis



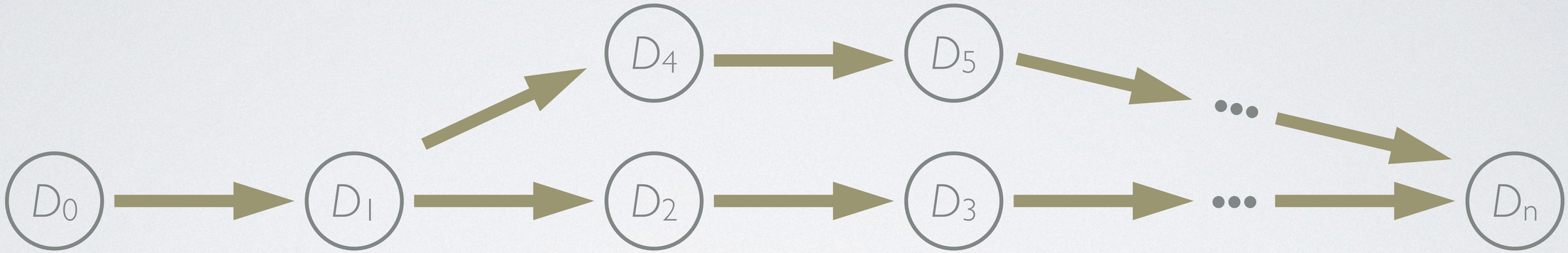
Type-Based Resource Analysis



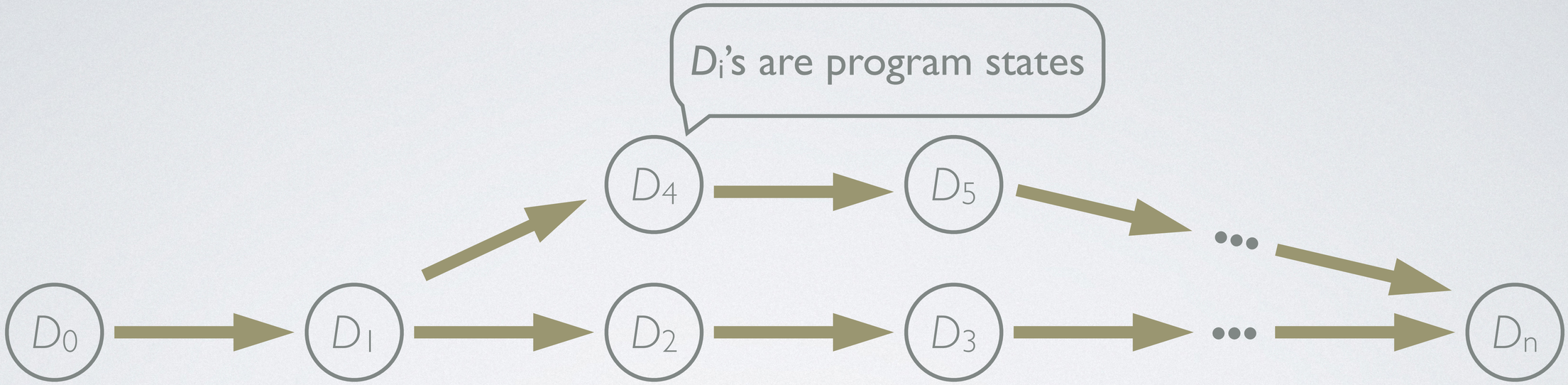
resource usage

The Potential Method

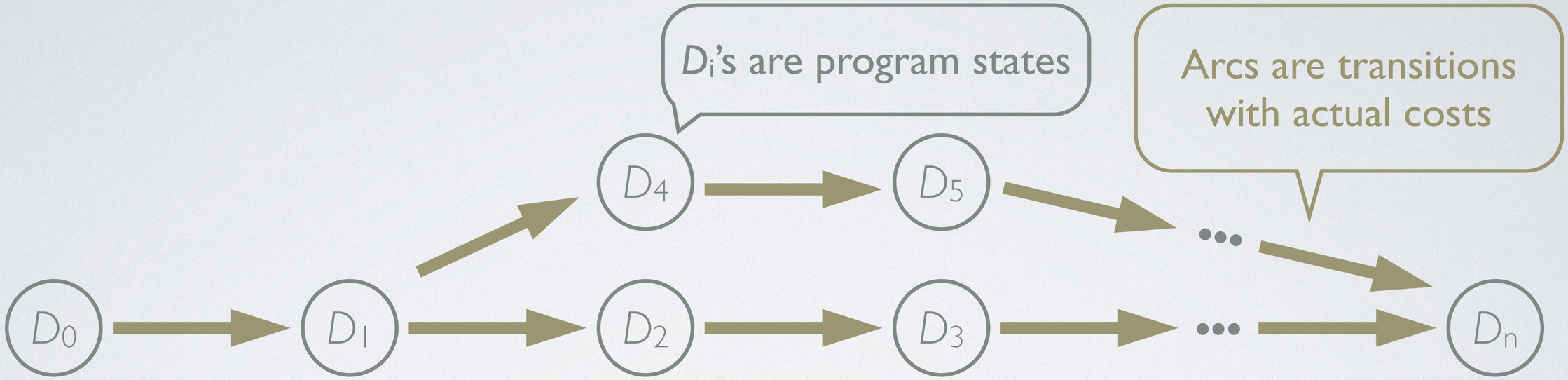
The Potential Method



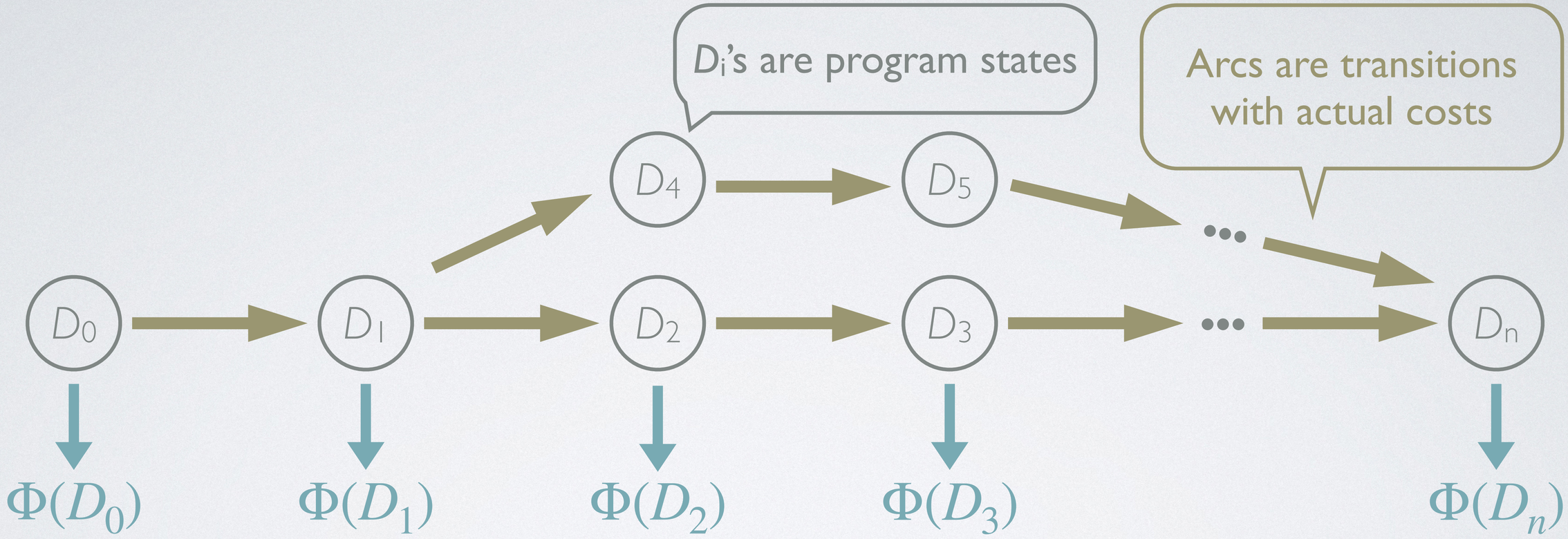
The Potential Method



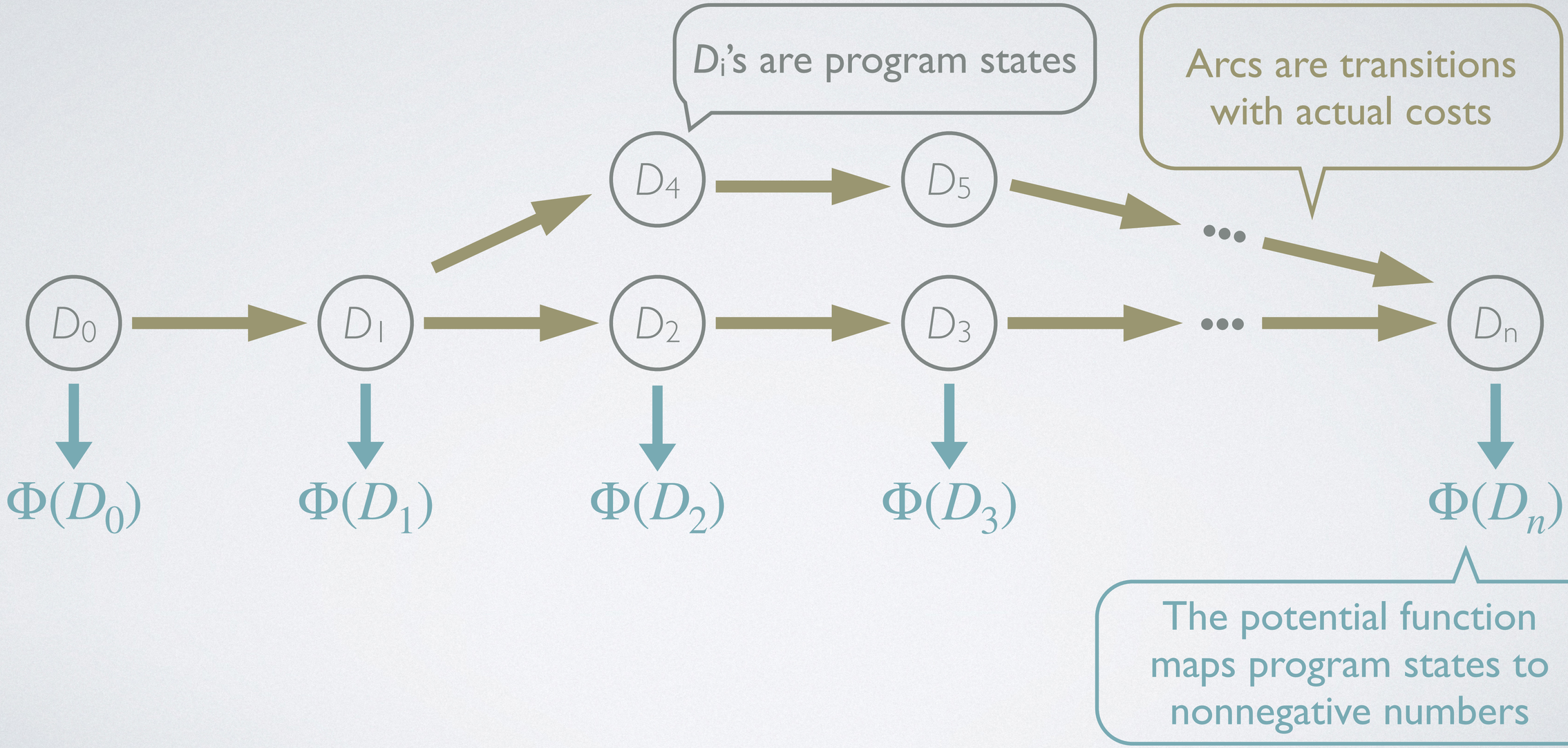
The Potential Method



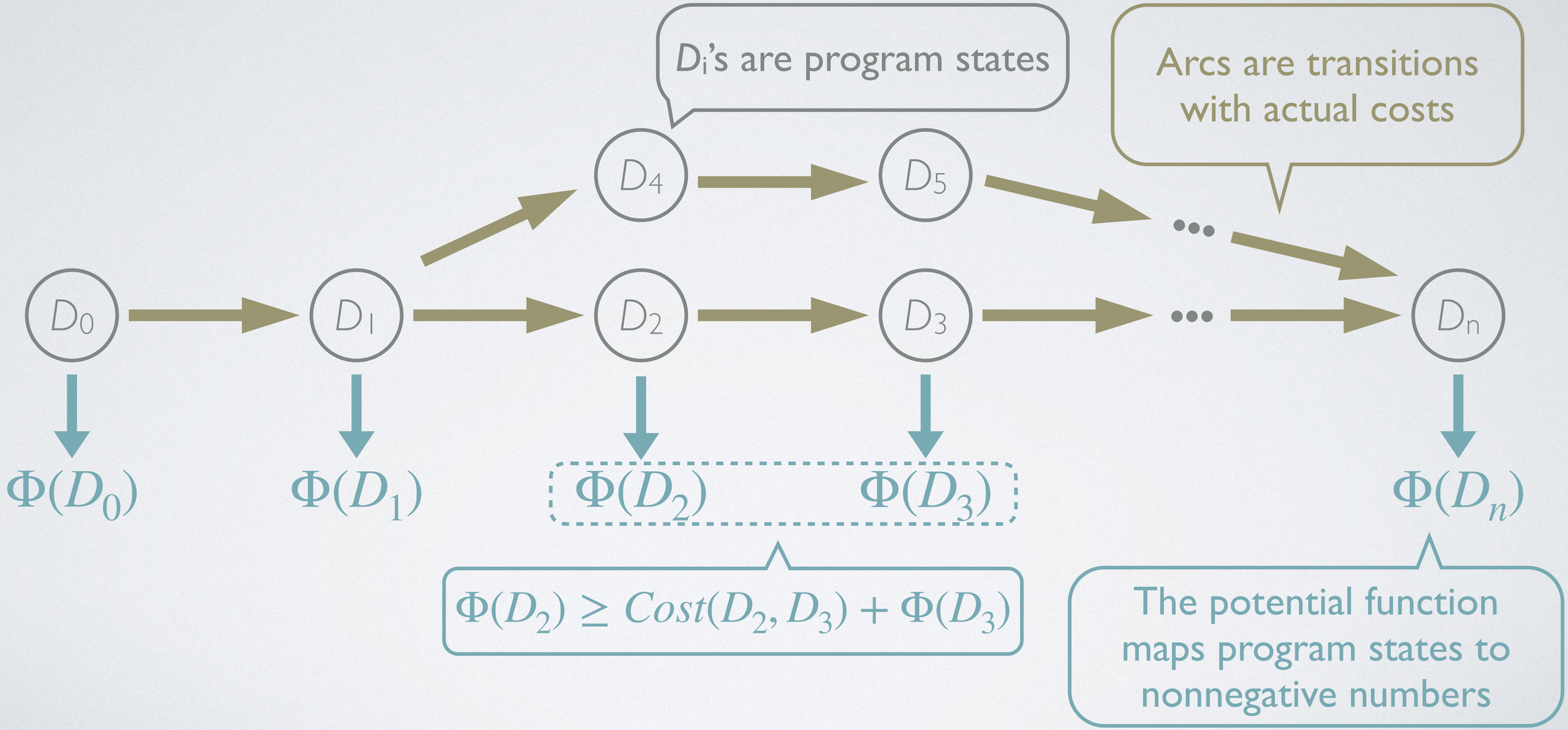
The Potential Method



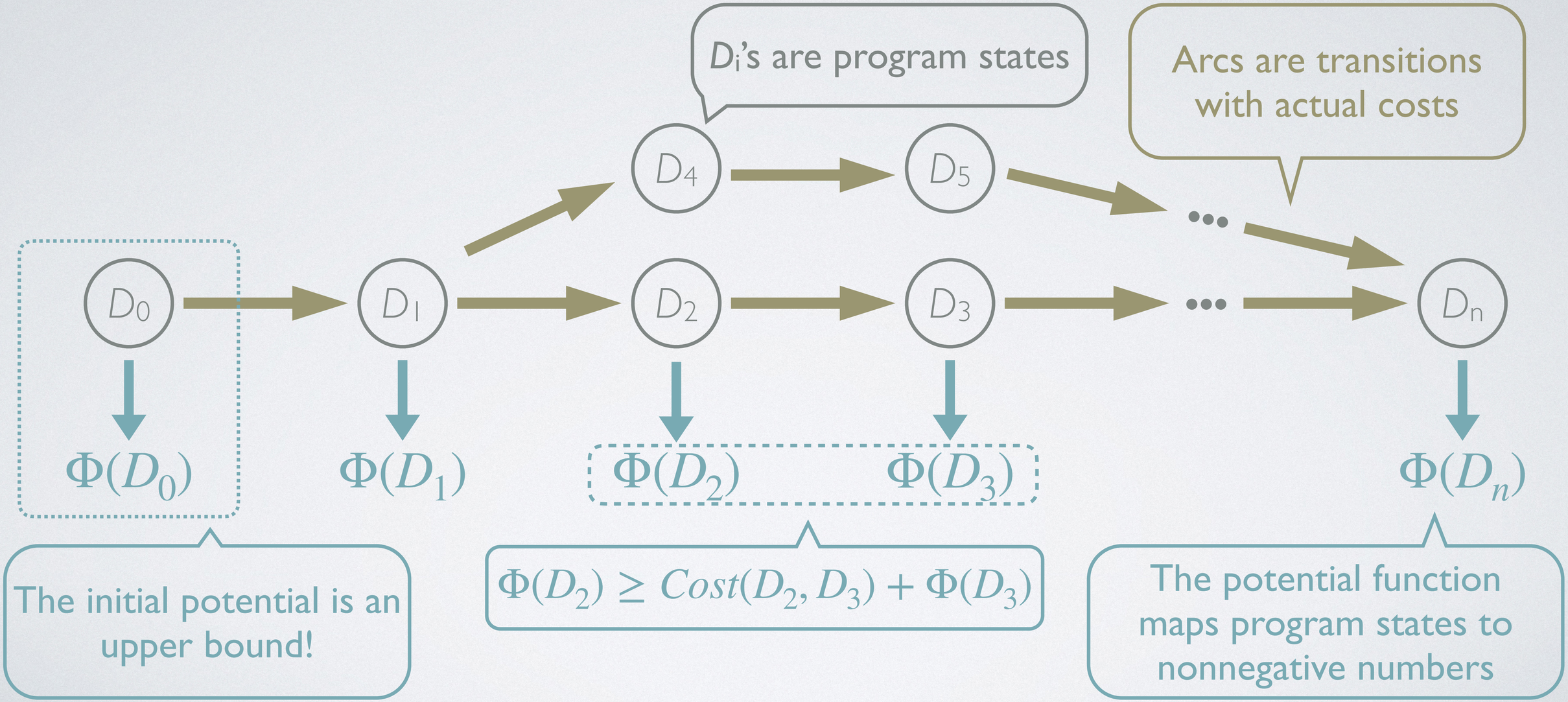
The Potential Method



The Potential Method



The Potential Method



Resource-Annotated Types

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

Resource-Annotated Types

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

Resource metric:
count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

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

Resource metric: count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

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

Resource metric: count function calls

$L^p(\text{int})$

Every element in the list carries `p` units of potential

Resource-Annotated Types

$Cost = |\ell_1|$

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

```
let rec append(l1, l2) =                    [l1: L¹(int), l2: L⁰(int)]; 0 units
  match l1 with
  | [] ->
    l2
  | x::xs ->
    tick(1);
    let rest = append(xs, l2) in
    x::rest
```

Resource metric: count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

`[l1: L¹(int), l2: L⁰(int)]; 0 units`

```
// l1 is consumed
```

Resource metric: count function calls

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
[l2: L⁰(int), x: int, xs: L¹(int)]; 1 unit
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
[l2: L⁰(int), x: int, xs: L¹(int)]; 1 unit
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

Resource metric: count function calls

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
[l2: L⁰(int), x: int, xs: L¹(int)]; 1 unit
[l2: L⁰(int), x: int, xs: L¹(int)]; 0 units
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

Resource metric: count function calls

```
let rec append(l1, l2) =                    [l1: L¹(int), l2: L⁰(int)]; 0 units
  match l1 with                             // l1 is consumed
  | [] ->                                   [l2: L⁰(int)]; 0 units
    l2                                      // l2 has the correct return type
  | x::xs ->                                [l2: L⁰(int), x: int, xs: L¹(int)]; 1 unit
    tick(1);                                [l2: L⁰(int), x: int, xs: L¹(int)]; 0 units
    let rest = append(xs, l2) in            [x: int, rest: L⁰(int)]; 0 units
    x::rest
```

Resource-Annotated Types

$Cost = |\ell_1|$

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

`Lᵖ(int)`

Every element in the list carries `p` units of potential

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

```
[l1: L¹(int), l2: L⁰(int)]; 0 units
// l1 is consumed
[l2: L⁰(int)]; 0 units
// l2 has the correct return type
[l2: L⁰(int), x: int, xs: L¹(int)]; 1 unit
[l2: L⁰(int), x: int, xs: L¹(int)]; 0 units
[x: int, rest: L⁰(int)]; 0 units
// x and rest are used to build a list
```

Resource metric: count function calls

Principle: The potential at a program point is defined by a static type annotation of data structures

Capability of the Potential Method

[RaML]	Multivariate polynomial bounds, amortized complexity, **push-button analysis**
[Atkey10]	Imperative programs, heap manipulation
[JHL+10]	Higher-order functions
[HM18]	Logarithmic bounds (e.g., splay trees)
[KH20]	Exponential bounds

[Atkey10] R. Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP'10*.
[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*.
[HM18] M. Hofmann and G. Moser. 2018. Analysis of Logarithmic Amortised Complexity. Available on: `https://arxiv.org/abs/1807.08242`.
[KH20] D. M. Kahn and J. Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *FoSSaCS'20*.

Overview

- [x] Motivation
- [x] Type-Based Resource Analysis
- [ ] Type-Guided Worst-Case Input Generation
- [ ] Evaluation

Overview

- Type-Guided Worst-Case Input Generation
  - Part I: Worst-Case Execution-Path Search
  - Part II: Type-Guided Pruning
  - Part III: Theoretical Guarantees
  - Part IV: Further Heuristics

WORST-CASE EXECUTION-PATH SEARCH

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

WORST-CASE EXECUTION-PATH SEARCH

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

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Symbolic Input

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Symbolic Input

$x_1 \mapsto z_1, xs \mapsto [z_2, z_3, z_4]$

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Symbolic Input

$x_1 \mapsto z_1, xs \mapsto [z_2, z_3, z_4]$

$x_1 \mapsto z_1, x_2 \mapsto z_2, xs' \mapsto [z_3, z_4]$

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Symbolic Input

$x_1 \mapsto z_1, xs \mapsto [z_2, z_3, z_4]$

$x_1 \mapsto z_1, x_2 \mapsto z_2, xs' \mapsto [z_3, z_4]$

When there is a **conditional** statement where we cannot decide which branch to take, we **try both branches** and record the **path conditions**

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$ Symbolic Input

$x_1 \mapsto z_1, xs \mapsto [z_2, z_3, z_4]$

$x_1 \mapsto z_1, x_2 \mapsto z_2, xs' \mapsto [z_3, z_4]$

When there is a **conditional** statement where we cannot decide which branch to take, we **try both branches** and record the **path conditions**

path condition: $(z_1 < z_2)$, resource usage: 2

Worst-Case Execution-Path Search

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$ Symbolic Input

$x_1 \mapsto z_1, xs \mapsto [z_2, z_3, z_4]$

$x_1 \mapsto z_1, x_2 \mapsto z_2, xs' \mapsto [z_3, z_4]$

When there is a **conditional** statement where we cannot decide which branch to take, we **try both branches** and record the **path conditions**

path condition: $(z_1 < z_2)$, resource usage: 2

path condition: $\neg(z_1 < z_2)$, resource usage: 0

Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages

$$\gamma \vdash e \Rightarrow \langle \psi, S \rangle$$

Symbolic Execution

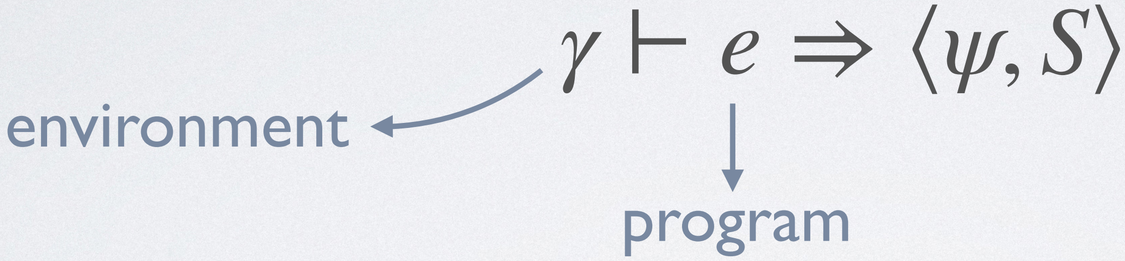
- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages

$$\gamma \vdash e \Rightarrow \langle \psi, S \rangle$$

environment

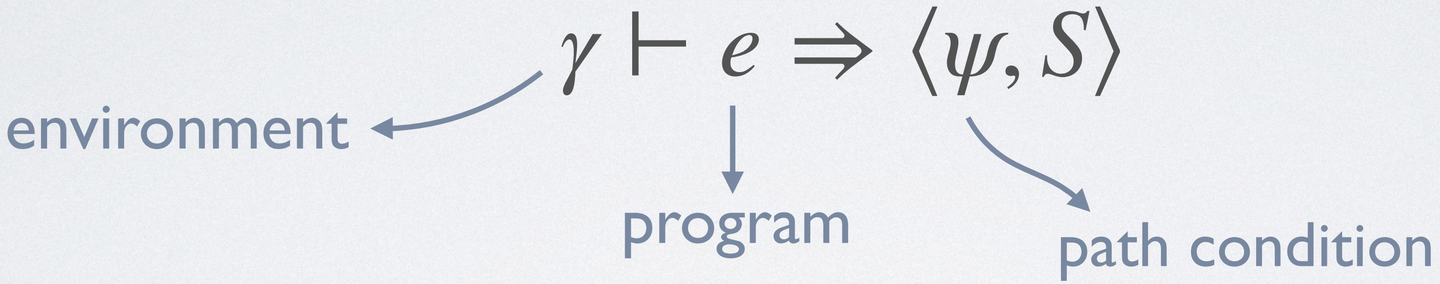
Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



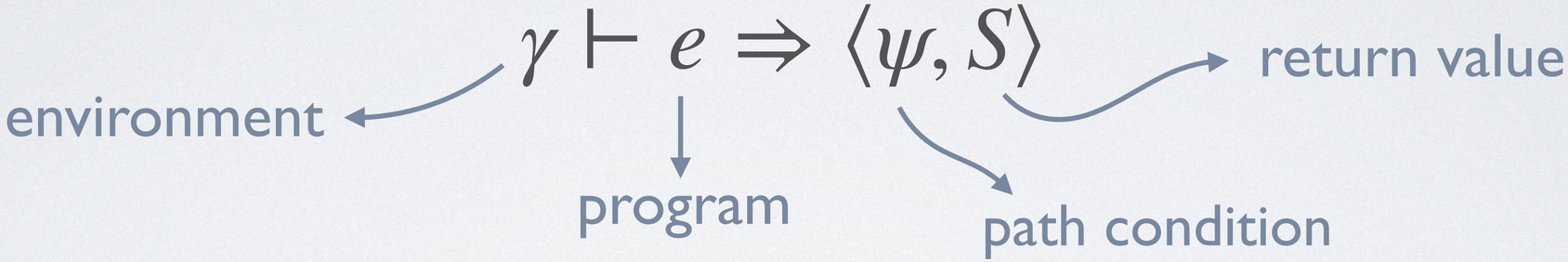
Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



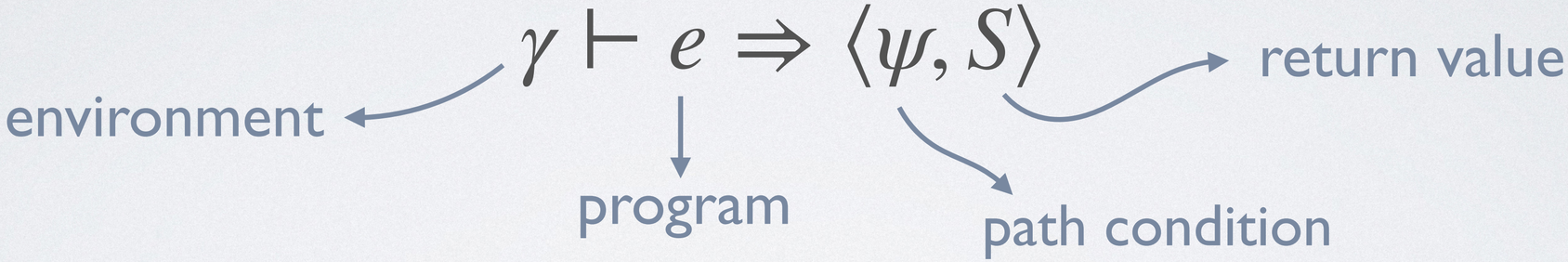
Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



Symbolic Execution

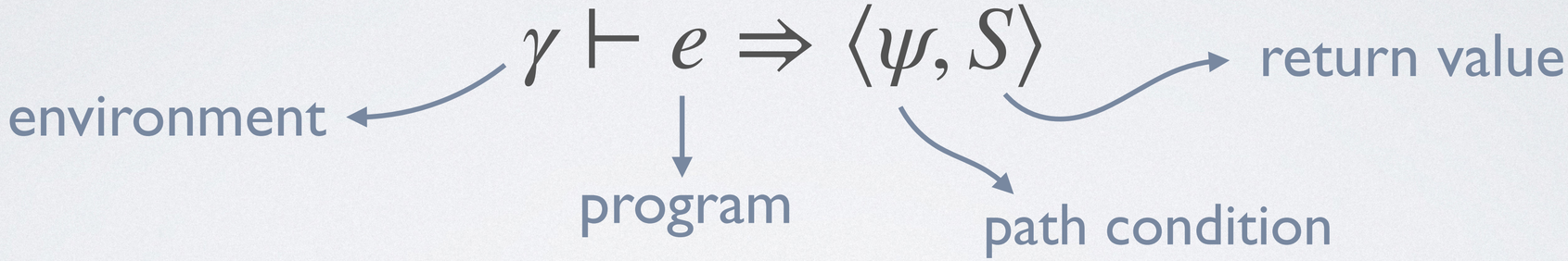
- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



$$x_1 \mapsto z_1, x_2 \mapsto z_2 \vdash (x_1, x_2) \Rightarrow \langle \mathsf{true}, (z_1, z_2) \rangle$$

Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages

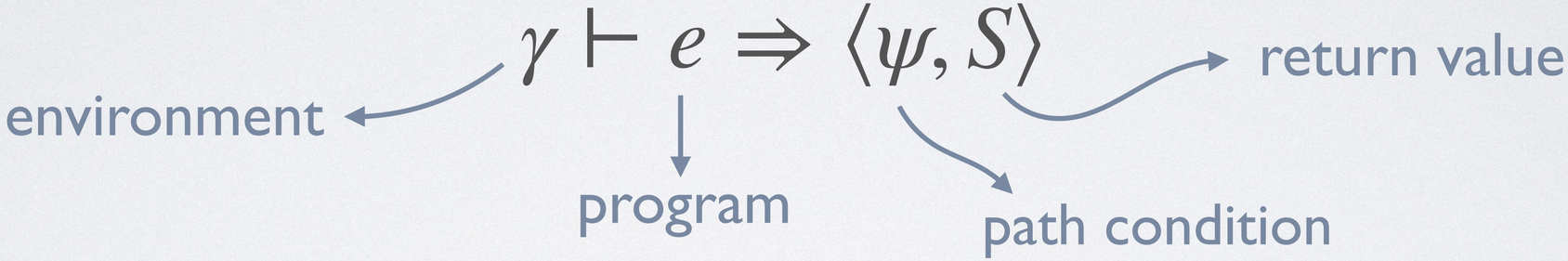


$$x_1 \mapsto z_1, x_2 \mapsto z_2 \vdash (x_1, x_2) \Rightarrow \langle \text{true}, (z_1, z_2) \rangle$$

$$x_1 \mapsto z_1, x_2 \mapsto z_2 \vdash \text{if } x_1 < x_2 \text{ then } (x_1 + x_2) \text{ else } (x_1 - x_2) \Rightarrow \langle z_1 < z_2, z_1 + z_2 \rangle$$

Symbolic Execution

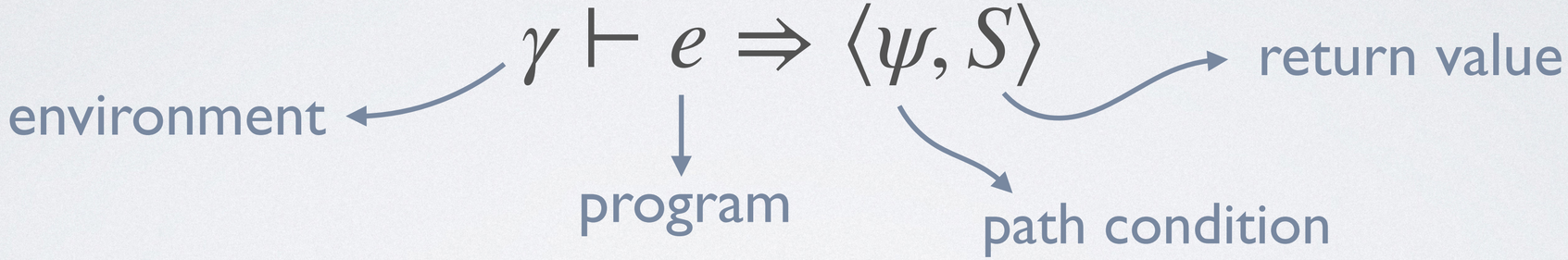
- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



- Symbolic execution rules for conditional statements:

Symbolic Execution

- **Idea: Enumerate** all execution paths, **record** path conditions, and **compare** resource usages



- Symbolic execution rules for conditional statements:

$$\textbf{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\textbf{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$ Resource usage

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$ Resource usage

$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$ 4

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$

	Resource usage
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$

	Resource usage
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 < z_4), [(z_3, z_4)] \rangle$	2

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \mathsf{lpairs}(\ell)$

	Resource usage
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 < z_4), [(z_3, z_4)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 \geq z_4), [] \rangle$	0

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \mathsf{lpairs}(\ell)$

	Resource usage
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 < z_4), [(z_3, z_4)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 \geq z_4), [] \rangle$	0

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$ — Resource usage

$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 < z_4), [(z_3, z_4)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 \geq z_4), [] \rangle$	0

- Use a constraint solver to find a model for the path condition, e.g., $z_1 = 0, z_2 = 1, z_3 = 0, z_4 = 1$

Symbolic Execution

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

Input specification: $\ell \mapsto [z_1, z_2, z_3, z_4]$

- There are four possible execution paths:

$\ell \mapsto [z_1, z_2, z_3, z_4] \vdash \text{lpairs}(\ell)$

	Resource usage
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 < z_4), [(z_1, z_2), (z_3, z_4)] \rangle$	4
$\Rightarrow \langle (z_1 < z_2) \wedge (z_3 \geq z_4), [(z_1, z_2)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 < z_4), [(z_3, z_4)] \rangle$	2
$\Rightarrow \langle (z_1 \geq z_2) \wedge (z_3 \geq z_4), [] \rangle$	0

- Use a constraint solver to find a model for the path condition, e.g., $z_1 = 0, z_2 = 1, z_3 = 0, z_4 = 1$
- A worst-case input for this function: $\ell \mapsto [0,1,0,1]$

Overview

- Type-Guided Worst-Case Input Generation
  - ☑ Part I: Worst-Case Execution-Path Search
  - Part II: Type-Guided Pruning
  - Part III: Theoretical Guarantees
  - Part IV: Further Heuristics

Type-Guided Pruning

- **Nondeterminism** leads to state explosion:

Then

$$\frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

Else

$$\frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Type-Guided Pruning

- **Nondeterminism** leads to state explosion:

$$\text{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \textsf{if } e \textsf{ then } e_1 \textsf{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\text{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \textsf{if } e \textsf{ then } e_1 \textsf{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Use the information about **potentials** obtained from **resource-annotated types** to **prune the search space** of symbolic execution

Type-Guided Symbolic Execution

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

Type-Guided Symbolic Execution

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

$\texttt{L}^{\texttt{p}}\texttt{(int)}$

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

Type-Guided Symbolic Execution

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

$\texttt{L}^{\texttt{p}}\texttt{(int)}$

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Type-Guided Symbolic Execution

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Type-Guided Symbolic Execution

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Type-Guided Symbolic Execution

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Type-Guided Symbolic Execution

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

`L^p(int)`

Every element in the list carries `p` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

$xs' : L^1(\text{int})$

Type-Guided Symbolic Execution

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

`L^P(int)`

Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

$xs' : L^1(\text{int})$

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

Type-Guided Symbolic Execution

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

`L^P(int)`

Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

$xs' : L^1(\text{int})$

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

Type-Guided Symbolic Execution

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

`L^P(int)`

Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Waste!

$xs' : L^1(\text{int})$

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

Type-Guided Symbolic Execution

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

`L^P(int)`

Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Waste!

$xs' : L^1(\text{int})$

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

If an execution path does not have **potential waste**, it must expose the worst-case resource usage

Type-Guided Symbolic Execution

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

`L^P(int)` — Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Waste!

$xs' : L^1(\text{int})$

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

If an execution path does not have **potential waste**, it must expose the worst-case resource usage

Prune the search space!

Type-Guided Symbolic Execution

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

$\mathtt{L^p(int)}$ — Every element in the list carries p units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Waste!

$xs' : L^1(\text{int})$

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

If an execution path does not have **potential waste**, it must expose the worst-case resource usage

Prune the search space!

In the paper, we present a full formalism of type-guided symbolic execution

Type-Guided Symbolic Execution

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

`L^P(int)` — Every element in the list carries `P` units of potential

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

$\ell \mapsto [z_1, z_2, z_3, z_4]$

$x_1 : \text{int}, x_2 : \text{int}, xs' : L^1(\text{int})$

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

$x_1 \mapsto z_1, x_2 \mapsto z_2,$

$xs' \mapsto [z_3, z_4]$

Cost = 2

Waste!

$xs' : L^1(\text{int})$

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

If an execution path does not have **potential waste**, it must expose the worst-case resource usage

Prune the search space!

In the paper, we present a full formalism of type-guided symbolic execution

$$\Phi; \gamma \vdash e \Rightarrow \langle \psi, S\rangle; \Phi'$$

Overview

- Type-Guided Worst-Case Input Generation
  - ☑ Part I: Worst-Case Execution-Path Search
  - ☑ Part II: Type-Guided Pruning
  - Part III: Theoretical Guarantees
  - Part IV: Further Heuristics

Theoretical Guarantees

Theoretical Guarantees

Soundness: If our algorithm generates ***an input***, then the input will cause the program to consume ***exactly*** the same amount of resource as the inferred ***upper bound***

Theoretical Guarantees

Soundness: If our algorithm generates ***an input***, then the input will cause the program to consume ***exactly*** the same amount of resource as the inferred ***upper bound***

Relative completeness: If there is an input of some given specification that causes the program to consume ***exactly*** the same amount of resource as the inferred ***upper bound***, then our algorithm is able to find ***a corresponding execution path***

Speed up Input Generation

Then

$$\frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

Else

$$\frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Speed up Input Generation

Then $\dfrac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$

Else $\dfrac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$

What if both branches do not waste potential?

Speed up Input Generation

- How about **eliminating** some generation rules?

$$\text{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\text{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

What if both branches do not waste potential?

Speed up Input Generation

- How about **eliminating** some generation rules?

$$\text{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\text{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

What if both branches do not waste potential?

Speed up Input Generation

- How about **eliminating** some generation rules?

$$\text{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\text{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Still Sound!

Speed up Input Generation

- How about **eliminating** some generation rules?

$$\text{Then}\quad \frac{\gamma \vdash e_1 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \gamma(e) \wedge \psi, S \rangle}$$

$$\text{Else}\quad \frac{\gamma \vdash e_2 \Rightarrow \langle \psi, S \rangle}{\gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \Rightarrow \langle \neg\gamma(e) \wedge \psi, S \rangle}$$

Still Sound!

- **Uniform-execution heuristic**: **Fix** the branch taken by each conditional statement before symbolic execution

Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?

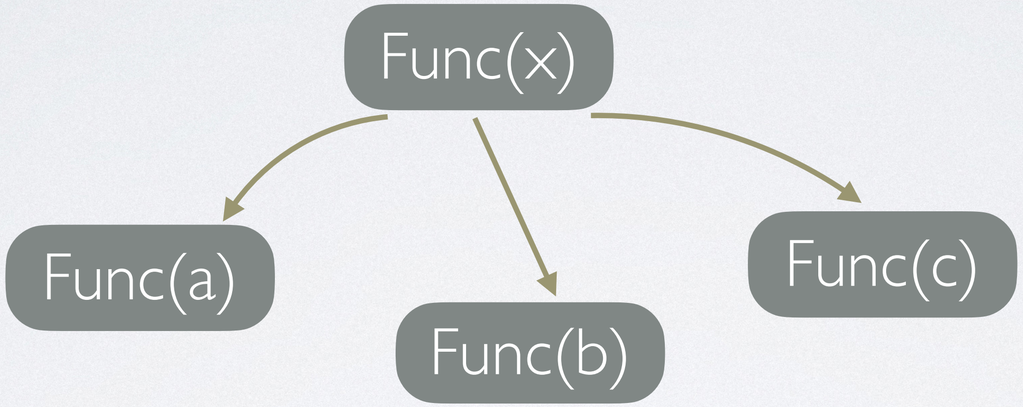
Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?

Func(x)

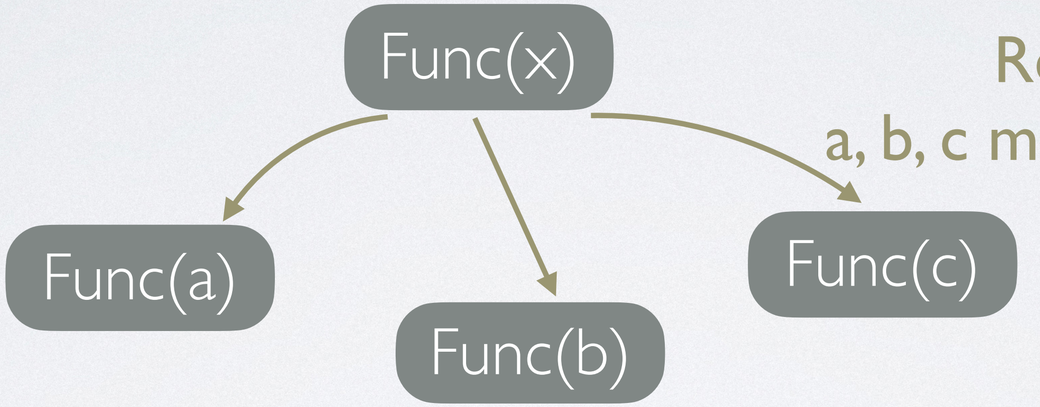
Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?



Compositional Input Generation

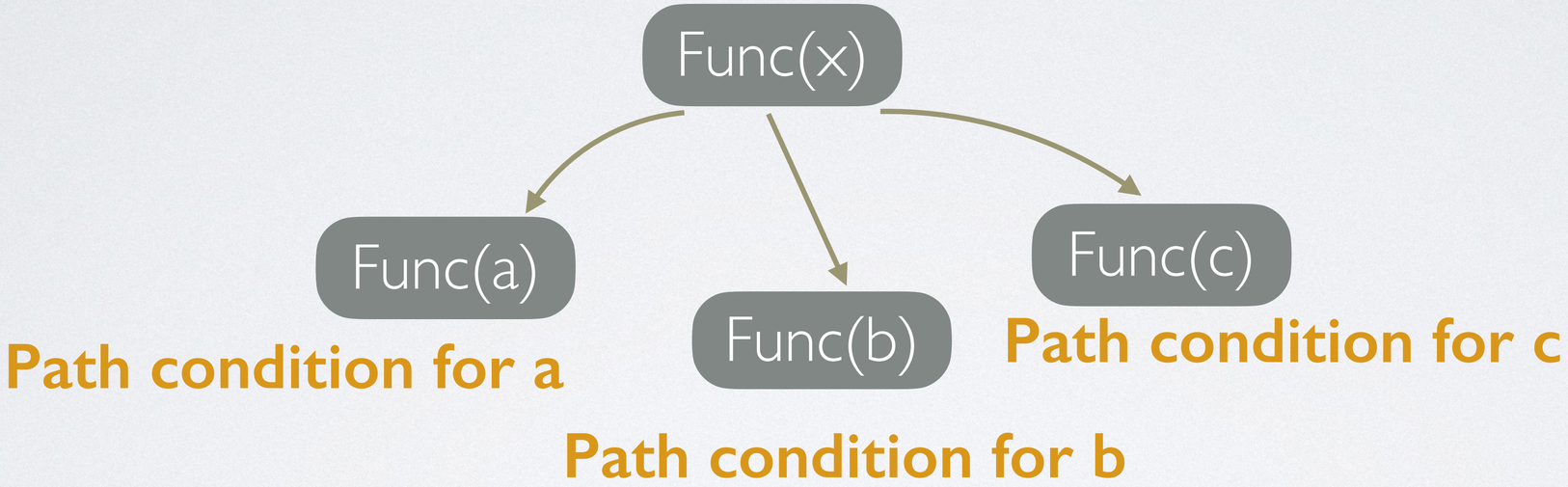
- Can we **generalize** the uniform-execution heuristic?



Recursive calls:
a, b, c might be subpart of x

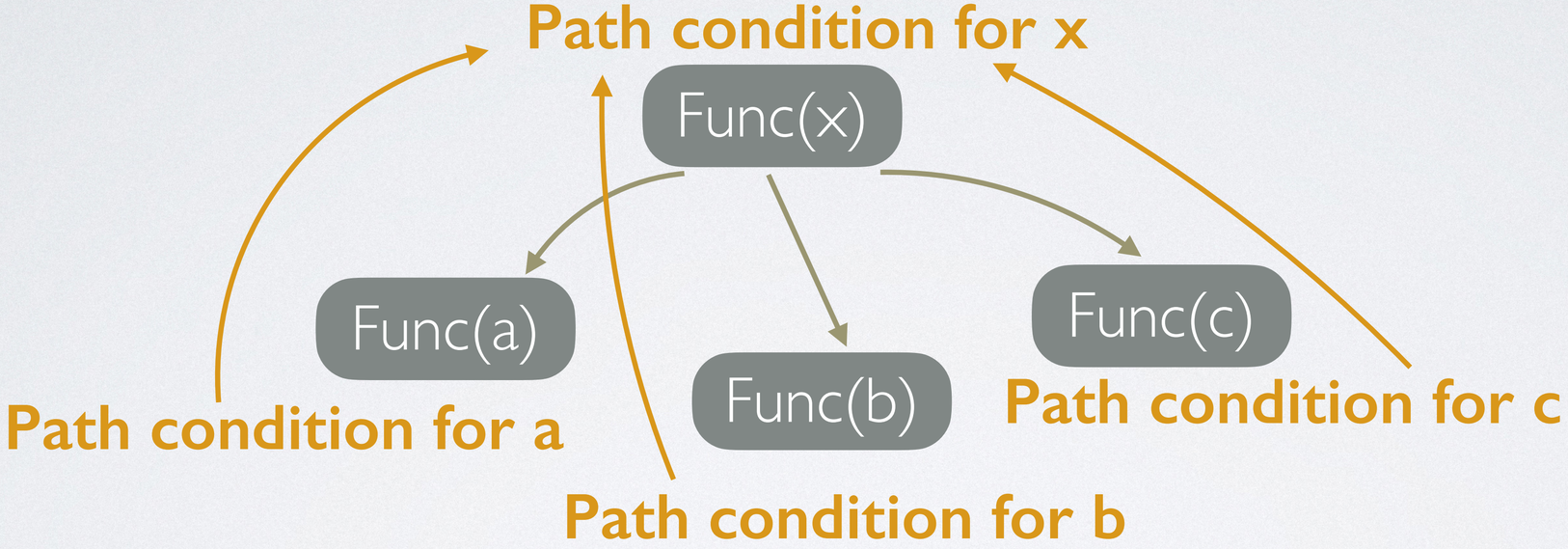
Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?



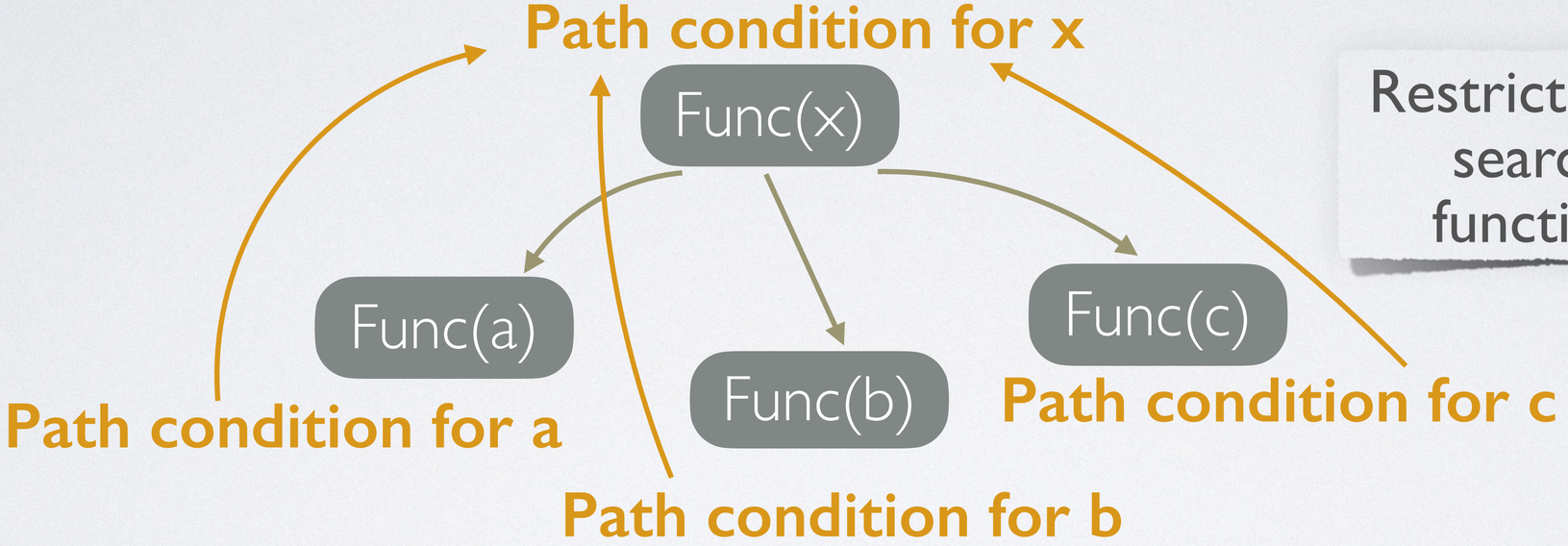
Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?



Compositional Input Generation

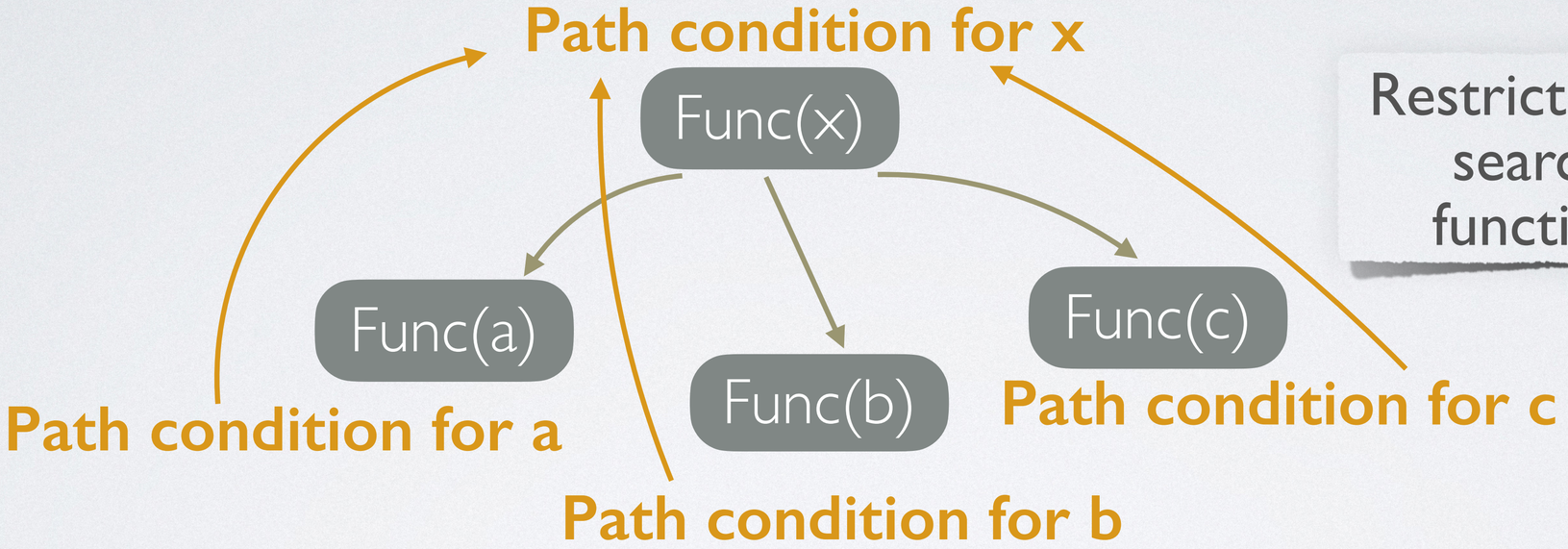
- Can we **generalize** the uniform-execution heuristic?



Restrict execution path search *inside* the function boundary

Compositional Input Generation

- Can we **generalize** the uniform-execution heuristic?



Restrict execution path search ***inside*** the function boundary

- Enforce all the calls with the ***same*** specification of inputs to execute the ***same*** path in the function body

Overview

- [x] Motivation
- [x] Type-Based Resource Analysis
- [x] Type-Guided Worst-Case Input Generation
- [ ] Evaluation

Implementation

- We implemented our generation algorithm for a purely functional fragment of **Resource Aware ML (RaML)**, including **higher-order functions, user-defined data structures**, and **polynomial resource bounds**

- We used the off-the-shelf constraint solver Z3 from Microsoft

Benchmarks (Selected)

Description	Shape	ALG	ALG+H1	ALG+H2
Insertion sort	200 integers	7.74s	**6.97s**	94.81s
Quicksort	200 integers	T/O	**53.23s**	157.21s
Lexicographic quicksort	Lists of length 100, 99, …, 1	439.35s	**438.79s**	T/O
Functional queue	200 operations	**444.64s**	T/O	T/O
Zigzag on a tree	200 internal nodes	T/O	T/O	**4.87s**
Hash table for 8-char strings	64 insertions	7.64s	**7.62s**	181.74s

Example: Hash Table

EXAMPLE: HASH TABLE

- Customized resource metric: count for **hash collisions**

Example: Hash Table

- Customized resource metric: count for **hash collisions**
- Use a hash function from a vulnerable PHP implementation

Example: Hash Table

- Customized resource metric: count for **hash collisions**
- Use a hash function from a vulnerable PHP implementation
- The program inserts 64 strings into an empty hash table

Example: Hash Table

- Customized resource metric: count for **hash collisions**
- Use a hash function from a vulnerable PHP implementation
- The program inserts 64 strings into an empty hash table
- Our algorithm "**realizes**" that it should find 64 strings with the same hash key, in order to trigger the most collisions

Comparison with Existing Approaches

Comparison with Existing Approaches

Dynamic

- Fuzz testing
- Dynamic worst-case analysis
- …

- Flexible & universal
- Potentially unsound: The resulting inputs might not expose the worst-case behavior

Comparison with Existing Approaches

Dynamic

- Fuzz testing
- Dynamic worst-case analysis
- …

- Flexible & universal
- Potentially unsound: The resulting inputs might not expose the worst-case behavior

Static

- Type systems
- Abstract interpretation
- …

- Sound upper bounds
- Potentially not tight: No concrete witness — the bound might be too conservative

Type-Guided Worst-Case Input Generation

The first provably correct worst-case input generation algorithm based on static resource analysis

TYPE-GUIDED WORST-CASE INPUT GENERATION

The first provably correct worst-case input generation algorithm based on static resource analysis

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

TYPE-GUIDED WORST-CASE INPUT GENERATION

The first **provably correct** **worst-case input** generation algorithm based on **static resource analysis**

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

- Integrated with RAML
- Effective on 22 benchmark programs

Experimental Results

Limitations:

- Purely functional programs
- Only work for tight bounds
- Depend on RAML

TYPE-GUIDED WORST-CASE INPUT GENERATION

The first **provably correct worst-case input** generation algorithm based on **static resource analysis**

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

- Integrated with RAML
- Effective on 22 benchmark programs

Experimental Results

Limitations:

- Purely functional programs
- Only work for tight bounds
- Depend on RAML

Future work:

- Support side effects
- Interact with resource analysis
- General theory for worst-case analysis

TYPE-GUIDED WORST-CASE INPUT GENERATION

The first **provably correct** **worst-case input** generation algorithm based on **static resource analysis**

- Formally developed algorithm
- Soundness & relative completeness

Theoretical Results

- Integrated with RAML
- Effective on 22 benchmark programs

Experimental Results